An Introduction to Embedded Programming in a Linux Environment

Richard Sevenich, Stuart Steiner, and Paul Schimpf

Copyright 2000 Lineo

Preface

This book is the result of collaboration between the sponsor, Lineo, Inc., and LinuxTeams. The book is intended for either a corporate training course or for an academic course. Although some of the chapters rely on a specific embedded target (the uCsimm with uClinux), we believe this allows students the hands on experience which is crucial to understanding. It is our observation that, for most students, understanding is achieved more easily by progressing from the specific case to the more general and abstract, than by starting from the general case. At the same time, we have not delved deeply into the hardware and have restricted our exposition to those hardware features that might be commonly found in today's embedded Linux systems. The hardware packaged with the course includes completely assembled circuitry allowing the uCsimm to support those features.

The first of the nine chapters consists of a terse introduction to Linux for those with little previous exposure to this operating system. Chapters 2 through 6 deal with the setup and use of an embedded Linux programming Environment. Chapters 7, 8, and 9 describe how to build an embedded Linux image. Chapter 7 does this more or less from scratch, but relies on existing open source resources, Busybox and Tinylogin. This material underlies that of Chapters 8 and 9 which demonstrate that an using embedix/SDK automates this process making it quicker and more robust.

This is a rapidly evolving area and therefore requires some disclaimers. The uCsimm form factor (30 pin SIMM) will likely be supplanted by a DIMM by the time this book is in use. Nevertheless, the SIMM makes for a simpler first exposure, which might actually be preferable in some circumstances. The uClinux software will certainly move to an enhanced version. We can expect to see related software for its various platforms improve and become readily available - particularly Real-Time extensions and remote cross platform debugging capabilities.

At this writing, the crucial Busybox and Tinylogin packages, which support Chapters 7, 8, and 9 are only at version numbers 0.47 and 0.78, respectively. We take some care here to explain in detail how to install them and work around their idiosyncracies. It would not be surprising to see them wrapped together and enhanced so that installation is more automatic. On the other hand, the current manual installation will leave the reader with a deeper understanding.

The version of Lineo's embedix/SDK available to us when Chapters 8 and 9 were written was a beta of version 1.2. It was still dependent on being used with the Caldera 2.3 Linux distribution. The intent is to ultimately make it Linux distribution neutral. That's not an easy goal in the infancy of the Linux Standard Base effort. Hopefully, it will soon be compatible with several of the common distributions. We made no attempt at the necessary modifications to do so at this point.

To the Instructor of a Corporate Training Course

As a corporate training course the book provides sufficient material for approximately a 30 hour course. The time required depends on the participant's familiarity with Linux and the C programming Language. It is suggested that as many of the Activities/Exercises be completed as possible. It would be best if each student had an individual setup consisting of a Linux workstation and the uCsimm/uClinux target hardware. That target is preassembled and includes

- the uClinux System Builder Kit CD
- the uCsimm 30 pin SIMM with resident uClinux
- the uCgardener board into which the uCsimm snaps in place
- an RJ-45 female ethernet connector (on the uCgardener) with an appropriate RJ-45 ethernet crossover cable
- an RS232 female DB9 serial connector (on the uCgardener) with an appropriate serial cable
- an appropriate power adapter which plugs into a power connector on the uCgardener
- 8 leds and 8 switches providing a byte of parallel I/O

The ethernet and parallel cables chosen are based on the assumption that the Linux workstation has the somewhat typical connectors i.e.

- an available serial port with a DB9 male connector
- an ethernet port with an RJ-45 female connector

It is also assumed that the Linux workstation has a CD ROM device for reading the uClinux System Builder Kit CD.

Chapters 8 and 9 require use of Caldera 2.3 as your workstation's Linux distribution. Further, to fully explore the capabilities, an extra/spare partition is

needed. Embedded images are written to either a floppy or the extra partition, as appropriate. The extra partition (or a Zip drive) is also used in Chapter 7.

As in any training course, time is of the essence and problems with the initial hardware/software configuration can be disastrous when their resolution is time consuming. Perhaps the most common pitfalls here would be

- the Linux workstation is not local network ready and cannot connect easily to the target [Note: it must have nfs server support configured]
- the Linux workstation is has connectors which are not compatible with the cables provided
- there are problems with the version of minicom on the Linux workstation

Hence the instuctor not only needs familiarity with the course material, but also needs to be confident that the training site is ready to go. This may require an extra setup day by the instructor or a liaison at the training site who can deal with potential setup problems.

To the Instructor of an Academic Course

Compared to the instructor of a corporate training course, your setup time is relatively relaxed. The material in the prior section aimed at the corporate trainer is appropriate for you, as well. However, although you need to get the system up and running, it need not be functioning properly when the student walks in the first day. In fact, setup problem solving by the student is good experience. Nevertheless, before the students appear, the hardware and software needs to be checked to a level where you are confident that any problems can be quickly resolved. This is, no doubt, part of your normal course preparation.

For an academic course, this material provides a basic introduction on which to build. The typical course might then add a project exploiting other capabilities of the uCsimm/uClinux combination. Ideas for such projects abound on the uCsimm email forum (ucsimm@uclinux.org) and a very realistic approach could require the students to search the forum for ideas and then make a formal proposal to the instructor. Alternatively, the instructor could take a lead role on project selection, particularly useful if there are hardware purchases involved (e.g. LCD panel with support hardware) and budget constraints to obey.

The material in Chapters 7, 8, and 9 lead to interesting projects as well. Here an embedded version based on a small image and root file system can be ported to a floppy disk. A larger, more feature rich ‘embedded’ system could be ported to a Zip

disk. For example, the floppy could lead to using a 486 rescued from the landfill to provide a simple router (there is an active Linux Router Project). With the Zip you could use a parallel port Zip drive as a somewhat portable ‘embedded’ system to couple to an otherwise abandoned PC to provide routing, firewall, etc. capabilities.

The chapters are intended to be covered in sequence with none skipped, except the first and only if the students are familiar with Linux. The activities/exercises are generally relatively easy, and can be supplemented by material intended as preparation for an added project.

To the student

You are aware of all the hype that surrounds Linux. The interesting thing about this phenomenon is that most of the hype is generated by enthusiastic users, rather than by the marketing departments of large corporations - although they have discovered the bandwagon. User enthusiasm arises because the system is powerful, usable, and because of its open source nature. The source is not only available, but individual users can get involved at whatever level is appropriate - even to the point of becoming kernel developers.

In this course, we explore Linux in the embedded systems world. It is most useful that Linux can be tailored to fit such targets. There is rising opinion that we are entering the ‘post PC’ era of personal, small embedded devices and that embedded Linux will play a central role. You are perhaps fortunate to have an opportunity to get involved in the early days of this new era.

This course introduces you to embedded Linux on a particular platform, the uCsimm/uClinux system. Nevertheless, most of what you will learn can be generalized beyond the hardware specifics. In Chapter 7 you’ll learn how to do it yourself with readily available hardware. In Chpaters 8 and 9, you’ll see that automating this process provides a much quicker and more robust development cycle.

If you are already an embedded developer, then your interest is in seeing whether embedded Linux will provide you with something new and useful. If you are a Linux user, new to embedded systems, your interest may lie in seeing how an embedded system can capitalize on a pared down Linux. If you are new to both embedded systems and Linux, you hope to accrue some initial expertise in each area. This course is intended to help all these constituencies move toward their goals. The course is introductory in nature and is a starting point rather than a final destination.

We encourage you to visit the uClinux website (http://www.uClinux.org) and there to join the uCsimm email list. There you will see many useful details discussed and be able to ask questions and contribute your own ideas.

Acknowledgement

The authors wish to thank Michael Angelo, Helen Chang, and Mark Steele, all of Lineo, Inc., for their interest, encouragement, and support. Similarly, we thank Laurel Helm, President of L.A. Boone Company, who conceived the LinuxTeams concept, making this type of project possible.

We are indebted to various reviewers particularly Jeff Dionne and Michael Leslie of Lineo and Dan Haffey and Greg Fortune of LinuxTeams.

Richard A. Sevenich, Ph.D.[1], Stuart Steiner[2], and Paul Schimpf, Ph.D.[3]

1. Department of Computer Science at Eastern Washington University and Consultant to LinuxTeams, rsevenic@netscape.net
2. Department of Computer Science at Eastern Washington University and Member of LinuxTeams, ssteiner@mail.ewu.edu
3. Department of Electrical Engineering at Washington State University and Consultant to LinuxTeams, pschimpf@foxinternet.net

Contents

Preface i

Contents vii

CHAPTER 1 *A Short Introduction to Linux* ***1***

Introduction **1**
What is Linux? **2**
Logging On **3**
The File System **4**
The File system Hierarchy Standard ***4***
The file system hierarchy in uClinux ***6***
Pathnames ***6***
File Permissions ***8***
Commands **9**
Some useful commands ***10***
Standard input, output, error ***15***
redirection and pipes ***16***
Editing Text Files **17**
Editing a file ***18***
Saving the file and/or quitting vi ***19***
Utilities **20**
gcc - the GNU C Compiler ***20***
gdb - the GNU Debugger ***21***
Activities/Exercises **27**
Logging on ***27***
Some Standard directories ***27***
The /proc directory ***27***
More about ls ***28***
mkdir and tree ***28***
tar ***28***
Use vi ***29***
The cp and mv commands ***29***
Using chmod, chown, chgrp ***29***
Using pipes ***29***

Using gdb ***30***
Using GNU make ***30***
References **31**

CHAPTER 2 *Overview of the uCsimm/uClinux Development Environment* ***33***

Introduction **33**
An Overview of the Development Environment **34**
An Overview of the Development Process **37**
Activities / Exercises **37**
Project Ideas ***37***
Minicom ***38***

CHAPTER 3 *Configuring the uCsimm/uClinux* ***39***

Introduction **39**
Configuring the Linux Workstation **40**
Installing the tool chains ***40***
Installation Glitches ***41***
Setting up the working environment ***42***
Configuring the uCsimm Target **44**
The RS232 Serial Connection ***44***
The Ethernet Connection ***44***
Getting the Serial Connection Up and Running **45**
Setting up minicom and the serial connection ***45***
Booting uClinux ***46***
Uploading a new OS image ***46***
Getting the Ethernet Connection Up and Running **48**
Configuring nfs on the Linux Workstation ***49***
Setting up the network parameters on your Linux workstation ***50***
Transferring uClinux images aided by nfs ***50***
Activities / Exercises **51**
Install the Tool Chains and a Working Directory ***51***
image.bin ***52***

Examining the src/ subdirectory in the uCsimm working directory ***52***
What steps are needed to recompile the uClinux kernel for producing a modified linux.bin? ***52***
Get the serial connection up and running ***53***
Create an image appropriate for establishing the subsequent ethernet connection ***53***
Get the ethernet connection up and running ***54***
Try out the flashloader command per Section 5.3 ***55***

CHAPTER 4 *The uCsimm/uClinux Resources* ***57***

Introduction **57**
Hardware Resources **58**
MCU Core Block ***59***
System Memory Block ***61***
Ethernet Controller Block ***62***
Software Resources **62**
The Bootloader ***62***
uClinux ***68***
Activities / Exercises **71**
Feature Overlap ***71***
bootloader commands ***71***
uClinux Extra/Replacement commands - inetd ***72***
uClinux Extra/Replacement commands - reset ***72***

CHAPTER 5 *Using the General Purpose Parallel I/O* ***73***

Introduction **73**
Two Example Programs using Port D I/O **74**
Example Program #1 - Write Port D ***74***
Example Program #2 - Read Port D ***76***
Running the Example Programs **79**
Running from a Testing Environment ***79***
Running in the FLASH ROM Environment ***80***
Primitives for an I/O Control Language **81**
The State Machine Engine ***81***
Broadening the State Engine Scope ***83***

Activities / Exercises **84**
Test the two example programs ***84***
Modify write_led.c ***84***
Incorporate a program into the FLASH ROM ***85***
Mini Project per Section 4 - a State Engine ***85***
An Example Using your State Engine ***85***

CHAPTER 6 *Using the uCsimm Ethernet Capability* ***87***

Introduction **87**
The Socket Mechanism using TCP/IP **88**
Pseudocode for a Simple Server ***89***
Pseudocode for a Simple Client ***89***
Supporting details for the pseudocode steps ***89***
A First Example - just the uCsimm **95**
The server program ***95***
The client program ***98***
Running the programs on the uCsimm ***100***
A Second Example - uCsimm and Workstation as client/server **101**
Changes to the server program ***102***
Changes to the client program ***102***
Running the server and client programs ***102***
Adding some Flexibility and Complexity to the Examples **103**
Setting socket options ***103***
Alternative to hard coded IP addresses ***103***
Serving Multiple Clients using Select ***104***
An example using select ***105***
Activities / Exercises **110**
Server and client on a single uCsimm ***110***
Server on Linux Workstation, client on uCsimm ***111***
Server on uCsimm, client on Linux Workstation ***111***
Change server to not care which of its IP addresses is used for the incoming connection ***111***
uCsimm to uCsimm ***111***
Using select - blocking ***111***
Using select - non blocking ***112***

References **112**

CHAPTER 7 *Making Your Own Embeddable Linux* ***113***

Introduction **113**
Busybox **115**
What is Busybox? ***115***
Getting Busybox ***120***
Tinylogin **121**
What is Tinylogin? ***121***
Getting Tinylogin ***121***
Compiling your new Kernel **121**
Small Linux on a Floppy **125**
Strategy - Floppy ***125***
Applying the Strategy - Floppy ***125***
Trying it out - Floppy ***131***
Small Linux on a Zip Disk or Extra Partition **131**
Strategy - Zip Disk or Extra Partition ***131***
Applying the Strategy - Zip Disk ***132***
Trying it out - Zip Disk ***140***
Revisiting uClinux **140**
Activities/Exercises **141**
Get Busybox ***141***
Create a new kernel image ***141***
Build a Small Linux on a Floppy ***141***
Build a Small Linux on a Zip Disk or on an Extra Partition ***142***
Investigating your zip system ***142***
Add networking capabilities ***142***
References **142**

CHAPTER 8 *Making Your Own Embedded Linux with Embedix SDK* ***145***

Introduction **145**
Installing Caldera 2.3 **146**
Requirements ***146***

One Hard Drive Containing Windows and Linux ***146***
Linux Only Hard Drive ***149***
Installing Embedix SDK **152**
Introduction to Target Wizard **153**
Starting Target Wizard ***153***
Creating A New Project ***155***
Loading A Kernel Configuration ***157***
Adding And Changing Components ***163***
Target Wizard Tutorials ***163***
Tutorials and Target Wizard Summary ***166***
Building and Installing Your Own Kernel **166**
Booting Into Your New Image **168**
Activities/Exercises **169**
Install Caldera 2.3 ***169***
Install Embedix SDK ***169***
Target Wizard Tutorial 1 ***169***
Target Wizard Tutorial 2 ***170***
Building and Installing Your New Kernel ***170***
Booting Into Your New Image ***170***
New Image Options ***170***
Different Target Wizard Options ***170***
Installing To a Floppy Disk ***170***
References **171**

CHAPTER 9 *Other Embedix SDK Topics* ***173***

Introduction **173**
Creating a Rescue Disk **174**
Saving Your Own State **175**
Embedded Linux with X-Windows **177**
Project -> Options -> Build Options Tab **178**
Build With Conflicts ***178***
Continue Building When Errors Occur ***179***
Merge User and Target Images ***179***
Build with NFS Client Support ***180***
Run LIPO on Image ***180***
Deploying A Target Image Using A Bootable Floppy Disk **180**

Frequently Asked Questions **182**
Activities/Exercises **184**
Creating A Rescue Disk ***184***
Saving Your Own State ***184***
Embedded Linux with X-Windows ***184***
Deploy A Target Image Using A Bootable Floppy Disk ***184***
Different Target Wizard/X-Windows Options ***185***
Minimal Embedded Image ***185***
References **185**
Contents i
Preface **ix**

CHAPTER 1

A Short Introduction to Linux

1 - 1 Introduction

It is the intent of this course to be practical. It is our experience that this requires hands on activities. Consequently, a choice of hardware is necessary and our choice has two elements:

- a Linux development station
- an embedded target running Linux

The Linux development station is generic, probably IA32 (Intel) based, but not necessarily so. If you already have a PC with Linux resident, that will suffice. For an embedded system, we have chosen the uCsimm (based on the Motorola DragonBall EZ) which is packaged with this course. Although this is a specific product, it works well for a general introduction. Most students learn efficiently by moving from specific cases to the more general and abstract. The uCsimm has many features, but this course explores only those we consider generic to embedded Linux. These features are

- an RS232 port for development
- parallel I/O
- an ethernet port

The readers of this book are expected to be reasonably adept C programmers with an interest in embedded Linux. It is possible that some will be unfamiliar with Linux and the purpose of this first chapter is to provide at least a rudimentary overview of how to use Linux. Those familiar with Linux might scan the chapter to see if they want to move directly to the next.

1 - 2 What is Linux?

Linux is a monolithic, multitasking, multiuser operating system (OS) which is a UNIX workalike. It is modularly designed with well understood interdependencies, which allows one to tailor Linux for a variety of purposes. In this course, Linux exists on the development workstation as a full blown OS with an incredibly large variety of features. However, it also exists on the target system as a very small, focused OS appropriate for an embedded system.

An exciting attribute of Linux is that it can be run on a wide variety of microprocessors and micro controllers. Once you are comfortable with Linux, you will find this to be a great advantage in the embedded systems arena. In particular, your development platform is a Linux box which can be used for many different embedded targets spanning a range of processor architectures, but all running Linux.

It is not the purpose of this introductory chapter to explore Linux in either great depth or breadth. The assumption is that the course participants are quite computer literate, but may have little previous experience with this particular OS. Hence our goal is to get up and running on a generic Linux box. To accomplish this the following sections will look at

- logging on to a Linux system
- the file system
- a selection of commands
- editing text files

Then we’ll close out the chapter with some activities/exercises followed by salient references.

1 - 3 Logging On

If you are a new Linux user and have your own machine, then you will be able to log in as root or as a user. Our assumption here is that you are logging in as a user. Working at root (also called super user or system administrator) level should be reserved for tasks that really require it e.g. many system administration tasks.

When Linux boots up, it either boots into the X Windows System (providing a GUI) or into console mode (no GUI). In either case, you will be prompted to log on. The logon is done in two steps in response to two successive prompts requiring that you:

- enter your username
- enter your password

So the system administrator must provide you a username and password ahead of time.

Upon successful logon, the OS starts a shell which responds to your input. There are many possible shells with the *bash* shell being the default shell for most Linux distributions. However, an embedded system is likely to use a more compact and less powerful shell. The shell interprets commands you enter at the command line, but also is a powerful scripting language. In particular, the shell can interpret text files which are written using the appropriate shell's grammar. It is beyond the scope of this course to investigate shell scripting. At any rate, once successfully logged in, you can enter commands and, in the X Windows environment, point and click on various icons.

Note: To log off, type *exit* at the command line. If you have your own personal Linux machine and need to power down, this must be done carefully. In particular, you need to be logged on with root privileges (don't *exit*) and then enter at the command line

```
shutdown -h now
```

There are variations on this (see *man shutdown*).

1 - 4 The File System

Linux, like most operating systems, has a hierarchical file structure. Application developers would like sufficient standardization in that hierarchy (and in other aspects of Linux) that they need not tailor their application's installation and operation for each style of Linux distribution. There is an evolving Linux Standard Base (see http://www.linuxbase.org/) and perhaps the most mature component is the file system hierarchy. However, the space and performance constraints of an embedded system will force an embedded Linux to stray from the standard. Nor is that particularly serious; an embedded system is special purpose system, less likely to cause an application developer pain.

In this section we'll look at the file system hierarchy standard and then at the hierarchy found in uClinux, as an example of an embedded system. Note that our embedded system development will typically be on the Linux workstation where the standard is hopefully followed. At the same time an understanding of the uClinux hierarchy will give us insight into the embedded Linux world.

1 - 4.1 The File system Hierarchy Standard

The *File system Hierarchy Standard*, which at this writing is at Version 2.1 can be found at http://www.pathname.com/fhs/. We'll give an overview of what we need for this course, but you are encouraged to look at the standard for the underlying rationale.

The root of the file hierarchy is symbolized simply by /. There is some confusion possible because one of the top level subdirectories under / is called *root*, but the latter is essentially a home directory for the root user (system administrator). The immediate subdirectories of / are described in the following table.

TABLE 1 - 1.

Directory Name	Directory Purpose/Contents
bin	essential command binaries which may be used by the system administrator and by ordinary users, required for system boot
boot	kernel image and configuration files used by the boot loader
dev	device files
etc	host-specific configuration files
home	user home directories
lib	essential shared libraries and kernel modules
mnt	mount point for temporarily mounting file systems such as those on a CDROM or floppy disk
opt	add-on application software packages
root	the root user's home directory
sbin	system binaries, essential for system administration, but not for system boot
tmp	location of temporary files
usr	secondary hierarchy, intended as shareable, read-only data. Here you'll find games, include files used by C programs, the linux source code hierarchy, and so on
var	variable data such as spool directories and log files

The *File system Hierarchy Standard* discusses each of these directories in some detail, not only the contents, but accepted conventions for their usage. Of course, the subdirectories contain files and further subdirectories and so on. Note that a directory is actually one of several kinds of files from the perspective of the OS, but it is convenient to make the distinction between directories and other files here.

All common distributions allow one additional directory at the same level as those of Table 1, the *proc* directory. It is not part of the standard, but deserves mention. It is often called a pseudo directory. It contains information about the executing kernel. The files hold data on such things as the resource usage of current processes,

what CPU is present, interrupt allocation, I/O port allocations, pci bus components, devices, modules currently installed, and so on. Much of the information is generated dynamically when it is accessed.

1 - 4.2 The file system hierarchy in uClinux

The directories at the top level of the unmodified uClinux hierarchy are tabulated next.

TABLE 1 - 2.

Directory Name	Directory Purpose/Contents
bin	per the standard
dev	per the standard
etc	per the standard
htdocs	when the uCsimm is accessed via ethernet using http, this is the directory accessed
lib	contains liberror.txt, a file containing text for a variety of error messages
proc	traditional
sbin	per the standard
tmp	a link to /var/tmp
usr	empty, suggested as the mount point for
var	presumably per the standard, but empty except for lost+found

As would be expected, these ‘standard/ directories are much sparser than a workstation resident Linux. Keep in mind that we do our development on our workstation rather than on the target. For example, if we write a C program to incorporate into our uCsimm/uClinux target, it is written and compiled on the workstation - so there is no need for the C program include files to be in the uClinux hierarchy.

1 - 4.3 Pathnames

If your username is, say, *jsmythe*, then when you log on you will be placed in jsmythe/ as your current working directory. Where is this located with respect to the root of the file hierarchy? The top level under root contains the subdirectories of Table 1, which includes the *home* subdirectory. The home directory contains all

user home directories including *jsmythe/* (the trailing slash indicates a directory, but may be omitted at the end of a pathname). Designating the root of the file system by a leading, solitary slash (*/*), we find the *jsmythe* log on directory by navigating like so

/ -> home -> jsmythe

This is compacted as a pathname

/home/jsmythe

If you used an editor to create the file *vita.txt* in this home directory, it would be found at

/home/jsmythe/vita.txt

Note: / is called the parent directory of *home*, *home/* is the parent directory of *jsmythe*, and *jsmythe/* is the parent directory of *vita.txt*.

A pathname that starts at the root of the file hierarchy starts with / and is called an absolute pathname. Clearly if you are 42 levels deep in the hierarchy, absolute pathnames become unwieldy. Hence, Linux also provides relative pathnames.

Let's look at some examples of relative pathnames. If you have logged on to your home directory at */home/jsmythe*, where you have created a file *vita.txt*, then its absolute pathname is, as before,

/home/jsmythe/vita.txt

but working from within the home directory /home/*jsmythe/*, you can refer to this file as

vita.txt or as ./vita.txt

The dot (.) indicates that the path starts at the current working directory; so, in our example, it effectively replaces */home/jsmythe*. Hence, when working within */home/jsmythe*, all three of the following are equivalent:

/home/jsmythe/vita.txt ./vita.txt vita.txt

Similarly the double dot (..) represents the parent directory of the current working directory.

1 - 4.4 File Permissions

File permissions are part of the security system. They help

- protect the OS from users, whether malicious or clumsy
- protect each user from other users

Permissions cover

- reading a file
- writing a file
- executing a file

Further, a file has three user categories

- the file's owner
- the file's group
- all other users

File permissions are written as three concatenated triads. The first triad refers to the file owner, and the second to the file group, and the third to all other user's. Here are some example permissions and what they mean:

> rwxrwxrwx => file owner, file group, and all others have read/write/execute permission.
>
> rw-rw---- => file owner and file group have read/write access, but all others have no access.
>
> rwxr-xr-x => file owner has read/write/execute access, while group and all others have read/execute access.

setuid and setgid programs

The three concatenated permission triads (for owner, group, and others) are preceded by another field whose purpose it is to give the user of the program special permissions. In particular, the user's effective uid (user ID) or effective gid (group ID) can be set to the file owner's uid or gid. As a somewhat dated example consider

a user changing his/her password in the days before shadow passwords became prevalent. The passwords were kept in /etc/passwd with permissions rw-r--r-- and root as owner. So the average user does not have write access to change a personal password. However, the /usr/bin/passwd command has permissions r-s--x--x where the s means that the user's effective uid is set to that of root and will be allowed to update /etc/passwd. The following commands can be used to set the user's effective id and effective gid to that of the file's owner

chmod u+s program ... we now say the program is a setuid program

chmod g+s program ... we now say the program is a setgid program

We'll see the *chmod* command in the next section.

1 - 5 Commands

Here we'll look at some useful commands which perform such tasks as file system manipulation and navigation, getting information, and modifying file permissions. Many of the commands care about permissions and the command will fail if you don't have the proper permissions. Linux, although relatively new, profits from the historical evolution of UNIX in that it comes close to achieving the following goals:

- there is a command or combination of commands to perform any task you can dream up
- the securely configured system prevents you from harming the system or other users

Part of the philosophy of Linux (again, inherited from UNIX) is that commands tend to be very simple, but can be combined very compactly to do complex tasks.

Note: Linux is case sensitive so Cat, CAT, and cat would be distinct - our typical commands will be lower case.

In this chapter, we'll consider some useful commands and then look at what might be called plumbing - standard input, standard output, standard error, redirection, and pipes. These plumbing operators are not commands, but rather operators provided by the shell. They are used in concert with the commands, allowing the commands to be used in combination.

1 - 5.1 Some useful commands

From the many commands provided by Linux, we'll consider those tabulated next.

TABLE 1 - 3.

Command	Purpose
ls	lists directory contents
cd	changes from the current working directory to another
pwd	show pathname of present working directory
mkdir	make a directory
rmdir	remove a directory (directory must be empty)
rm	removes a file, but has a recursive option that can be used to remove non empty directories as well
cp	copies a file
mv	moves its argument (a file name) to a new name
more	lists its argument a page at a time
cat	concatenates its arguments and sends the result to standard out
grep	print lines matching a pattern (regular expression)
tar	archives its argument (with compression if specified)
man	displays the manual pages of its argument
su	switch to another user
chmod	changes file/directory permissions
chown	changes file/directory owner
chgrp	changes file/directory group

We'll discuss these only briefly, and in most cases they have many further options which can be explored via the man pages.

The ls command

If you want to list the contents of your current working directory, you can simply type

```
ls
```

and if you want lots of information (e.g., permissions, sizes), use the long form

ls -l

As you would expect, you can list contents of other directories (as allowed by the permissions) e.g.

ls -l /usr/src

Other options allow sorting on such information as date etc.

The cd command

This allows you to change your working directory to another, if permissions allow. Here are some examples:

cd / => change to the root directory of the file system

cd /usr/src => change to /usr/src

cd .. => change to the parent of your current directory, i.e. one level up

cd ../.. => change two levels up

cd - => change back to the directory you were in before this one

The pwd command

This is the ‘present working directory’ command. It displays the absolute pathname of where in the directory hierarchy you currently are.

The mkdir and rmdir commands

These can make and remove (empty) directories. Assuming our current working directory is /home/jsmythe, consider these sequential examples:

mkdir emails => creates the directory /home/jsmythe/emails

mkdir emails/outgoing => creates directory /home/jsmythe/emails/outgoing

rmdir emails/outgoing => removes directory /home/jsmythe/emails/outgoing

The rm command

This command allows you to delete files. For example, if you are in directory */home/jsmythe* and want to delete a file at that level named *vita.txt*, you could enter

```
rm vita.txt
```

This is a very powerful command in that you can delete a whole chunk of a hierarchy. For example, if */home/jsmythe/project1/* was the top of a significant hierarchy of subdirectories and files, you can delete it via

```
rm -rf /home/jsmythe/project1
```

The cp and mv command

If you want a copy of a program, the format is

```
cp existing new
```

for example,

```
cp vita.txt vita1.txt
```

Now there are two files, the original (*vita.txt*) and a new copy (*vita1.txt*).

On the other hand, if you want to rename a file, the syntax is

```
mv existing new
```

for example

```
mv vita.txt resume'.txt
```

In this case, the file *vita.txt* has been replaced by the identical file, except that it has a new name, *resume'.txt*.

The *mv* command can also be used to rename any node in the file hierarchy e.g.

```
mv project/ project1/
```

The more command

This command can display a file at the console, one screen full at a time. For example,

```
more vita.txt
```

Once invoked, more provides some helpful key strokes:

- space key => display next screen
- q key => quit

We'll see later how this command becomes more powerful with the capabilities explored in section 5.2.

Note: a similar, but more powerful command is *less*.

The cat command

The *cat* command concatenates its arguments and displays the entire result at the console, so a multi-screen result will quickly scroll off the end. For example,

```
cat file1 file2
```

As with the prior command, we'll see later how this command becomes more powerful with the capabilities explored in section 5.2.

The grep command

The *grep* command can search a file or set of files for a pattern. For example, let's say we're searching for mention of *sys_write* in any of the C files in a directory. This could be done via

```
grep 'sys_write' *.c
```

Then *grep* would print all lines in all files mentioning the sought for pattern, *sys_write*. There is also a *recursive grep* (*rgrep*) available which can search recursively through a directory hierarchy.

While we're on the subject of searching, there is a wonderful tool for searching through the source hierarchy which can be used over the web or downloaded. See http//lxr.linux.no/.

The tar command

We'll just give two examples of this convenient and powerful command. Let's say you have a project at work which you also pursue at home and are faced with moving the project back and forth as you work on it in both places. Let's say the project hierarchy starts at the node */home/jsmythe/project/* and consists of many subdirectories and files. To compress and archive the hierarchy into a single file called *proj.tgz*, use the command

```
tar cvfz proj.tgz project/
```

where we have assumed that the command is issued from directory */home/jsmythe*.

To subsequently expand and unarchive the file on your other machine from directory */home/jsmythe*, use

```
tar xvfz proj.tgz
```

and it will produce the /home/jsmythe/project hierarchy if it does not currently exist and append to it if it does.

Note: Linux provides a powerful tool called *cvs* for help in managing projects which have multiple developers.

The man command

If a manual page on some entity, *name*, exists, then

```
man name
```

will give the information. For example, if you want to learn what options are supported for the *ls* command, enter

```
man ls
```

or to learn about the *vi* editor, enter

```
man vi
```

You can even learn more about *man* by entering *man man.*

The su command

The switch user command (sometimes misnamed the super user command) allows you to switch to a different user. If no argument is supplied, you will switch to super user if you successfully respond to the subsequent prompt for the super user password. If an argument (a valid username) is provided then you will switch to that user if you successfully respond to the subsequent prompt for that user's password. You return to your previous identity with the *exit* command. As you can see, it is essentially a nested log on scenario.

The chmod, chown, and chgrp commands

These allow you to change permissions (*chmod*), ownership (*chown*), and group (*chgrp*) of files to which you have appropriate access. There are typically used for system administration. For example, as root I could change a file's owner and group to fred_smith (assuming there is such a user and group) via

```
chown fred_smith.fred_smith filename
```

Similarly, if an owner wants to change a file's permissions to rw-rw-r--, the appropriate command would be

```
chmod 664 filename
```

For those uncomfortable with octal permissions (like 664 representing rw-rw-r--), there is the option of using letter based syntax. Investigate these commands further via their man pages.

1 - 5.2 Standard input, output, error

Commands and other programs often expect input to come from the keyboard, which we call standard input. Similarly they often send output to the console, which we call standard output. Error information is also typically sent to the console, but the destination is then called standard error. We'll see in the next section that the distinction between standard output and standard error is not as trivial as it first seems. A command such as

more vita.txt

sends its output to standard output, the console.

1 - 5.3 redirection and pipes

The simple concepts of standard input, output, and error become significantly more powerful when used with redirection and pipes. Some examples are in order.

redirection

Recall that the command

cat file1 file2

will concatenate file1 and file2, sending the output to standard out, the console. However, standard output can be redirected to a file other than the console e.g.

cat file1 file2 > file3

where the right arrow (>) is the redirection operator for standard output. In this case file3 will hold the concatenated result of file1 and file2. In the case just shown file3 is created and any existing file by that name would have been destroyed. If file3 already existed and you wanted to append the new information, you would enter

cat file1 file2 >> file3

where the double right arrow is called the nondestructive redirection operator.

Similarly, the left arrow (<)is the redirection operator for standard input. There is also a syntax (somewhat shell dependent) for redirecting standard error. For example, in the bash shell we might perform a compilation with this syntax:

gcc file5.c -o file5.c 2> file5.err

where the last piece *2> file5.err* redirects standard error (but not standard output) to the file called *file5.err.* This is quite useful when your compilation triggers many errors which scroll down the screen. Typically, the early errors are significant, but may scroll off the screen. However, they will be available in a redirected error file such as the example's *file5.err.*

These redirection operators appear very handy, but when combined with the pipe concept the power is much enhanced.

the pipe

The pipe connects commands so the output of one becomes input for the second. Let's look at its use without redirection, first. Consider using *ls -l* on a very large directory. It would scroll off the screen. However you can pipe it to *more*, which you'll recall, displays one screen at a time. The syntax is

```
ls -l | more
```

where the vertical bar (|) is the pipe operator. You can pipe several commands together and combine them with redirection. Here is a slightly more involved example,

```
cat file1 file2 | sort > file3
```

which does these tasks in order

- concatenates file1 and file2
- sends the result to the *sort* command, which sorts the lines in alphabetical order
- stores the alphabetized, concatenated result as a new file called file3

The above set of tasks might be given to beginning programming students who would then write some high level language program to get the result. Here we see how the UNIX philosophy of simple commands used in combination can do complex tasks.

1 - 6 Editing Text Files

Since we'll be programming, we need a text editor. There are many available with Linux and programmers become emotionally attached to their preferred editors. However, the one editor that will always be available is *vi*. In particular, if an embedded Linux has an editor, it is most likely *vi*. For example, uClinux provides *vi*. Consequently, we'll give a quick overview of *vi*, with no insistence that it is the best editor, merely that it is omnipresent. The references at the end of this chapter give more thorough expositions. Also note that GNU version of *vi* (called *vim*, but will answer to *vi*) provides a tutorial. Further, it will operate in what you might

consider to be a more friendly fashion than classic *vi*. If *vim* is available on your system with its documentation, you can run the tutorial by entering

```
cd /usr/doc/vim-common-5.6/tutor

vi tutor
```

where the version number (5.6) may be different on your system.

If you haven't encountered a modal editor before, vi will seem strange at first (maybe later on, too). It provides three modes of which we'll discuss only the first two:

- command mode
- insert mode

Upon startup, vi is in command mode. Our approach here will be demonstrate how to edit a file, save it, close it, and then reopen it for subsequent editing. We will not explore *vi* in any depth.

1 - 6.1 Editing a file

To edit a file (let's say *hello.c*), invoke *vi* at the command line as follows:

```
vi hello.c
```

If the file does not yet exist, this will allow you create, edit, and then save the file with this name. If the file does already exist, it will be opened for editing with *vi*.

As mentioned earlier, *vi* wakes up in command mode, but we want to enter text and to do so we enter a command which puts us into insert mode where we can enter text. The command is simply to press the *i* (for *insert*) key. You can now enter text. The *vim* implementation allows you to delete characters with the backspace key and navigate with the arrow keys while in insert mode.

Classic *vi* is less friendly, requiring that you return to command mode for character deletion and navigation - so the insert mode is for text insertion only. To return to command mode press the *Esc* key. To then navigate in classic *vi*, one uses these keys to move the cursor:

- *h* moves cursor left
- *j* moves cursor down

- *k* moves cursor up
- *l* moves cursor right

For text deletion in classic vi, we have these keys:

- *x* deletes the character under the cursor
- *dw* deletes the word the cursor is on
- *dd* deletes the line the cursor is on

Note that vim can perform its deletion and navigation tasks in the classic fashion as well as in its more contemporary style.

The upshot is that editing with vim can be done remaining in insert mode; whereas editing with classic vi requires constant moving back and forth between command and insert modes.

The current uClinux comes with the classic vi. Your Linux workstation likely has a variety of editors and the implementation of vi is probably vim, but if you're not a vi devotee you'll possibly prefer a different editor. However, vi is much more powerful (many more commands etc.) than our terse description suggests. Other editors, each with its own following, include

- emacs - present in typical Linux distributions (really more than an editor)
- nedit - http://www.nedit.org
- cooledit - http://cooledit.sourceforge.net/

plus many more.

We'll later see that you can do virtually all of your editing tasks on the Linux workstation because your working directory can be nfs mounted on the target uClinux file system hierarchy. Nevertheless, there might be an instance where the classic vi on uClinux is what you need.

1 - 6.2 Saving the file and/or quitting vi

Remember:

- when in command mode, you enter insert mode by pressing the *i* key
- when in insert mode, you enter command mode by pressing the *Esc* key

Saving and quitting require that vi be in command mode. Each of these commands begins with a colon. Here are the possibilities:

:q will quit *vi*, but only if the file is unmodified - the file will not be saved by this command

:q! will quit *vi* whether or not the file is modified - the file will not be saved by this command

:w will save the file, but not quit *vi*

:wq saves the file and quits *vi*

1 - 7 Utilities

There is perhaps no true distinction between a command and a utility, but when a command becomes quite powerful and feature rich, it seems appropriate to distinguish it form a simpler command and call it a utility. Certainly *gcc*, *gdb*, and *make* fall into this category. These will commonly be used by anyone doing development at kernel level. There are, of course, others beyond the scope of this book e.g. *cvs, lex, and yacc*. At any rate, those we mention here all have large manuals available form the Free Software Foundation.

1 - 7.1 gcc - the GNU C Compiler

The *gcc* manual is deservedly large. There is no way we can discuss its capabilities here, but we will give some example command line invocations which you can use to identify a few areas to explore:

gcc file.c -o file.x ... this compiles source file file.c and creates an executable named file.x

gcc -c file1.c ... this compiles source file file1.c and creates an object file named file1.o

gcc -o app1 main.o file1.o file3.o .. this links three object files (main.o, file1.o, and file3.o) into a single executable named app1

gcc -S file2.c ... this compiles source file file2.c down to assembly language, creating a file named file2.s (Note that the command line S is upper case and the resulting assembly language file suffix is a lower case s)

gcc -g file4.c -o file4 ... this compiles source file4.c passing along appropriate symbol information to support running the resulting executable (file4) under control of *gdb*, the GNU debugger

1 - 7.2 gdb - the GNU Debugger

The *gdb* manual is also large. Perhaps the best approach here is to compile a C source file with debugger support and then dive in with *gdb.* You can set break points, single step, examine processor registers, program variables, set values, and so on.

Let's say we want to run a program under *gdb* control. For a source program example.c we could compile it and then invoke *gdb* to run the executable e.g.

gcc -c example.c -o example

gdb example

One then gets the *gdb* prompt, at which you could enter help to start an investigation of *gdb* capabilities i.e.

(gdb) help

For an embedded Linux developer, the attraction of *gdb* lies in the fact that one can debug remotely via the serial port or ethernet and it can be done cross platform. These already existing capabilities take some up front work to deploy. Because of the heightened interest in embedded Linux, we can expect a spate of articles etc. detailing how to do such cross platform, remote debugging.

We should mention that *gdb* is a command line creature, but there exist graphical front ends. Perhaps *ddd* is the most widely used of these.

1 - 7.3 make

The *make* command is actually a powerful utility. It can be used to generate a series of commands to be executed by the shell. It is intended for use in maintaining files and, most commonly, for generating the final executable for a complex collection of

programs. The *GNU Make* documentation is cited in the *References* section, but many Unix programming books will also have a chapter or section on *make*. There is an excellent O'Reilly book on the System V Release 4 version of *make*, but here we stay with *GNU make*.

Since *make* works with a potentially large set of interdependent files, we need to specify those interdependencies along with what we want done. This information is provided in a *makefile*. If *make* is invoked without specifying a *makefile* it works through the default names in order until it finds one. These names are

- GNUmakefile
- makefile
- Makefile

However, you can specify your own choice e.g.

```
make -f my_makefile
```

We cannot do any more here than hint at the power of *make*. Nevertheless, we'll find it useful to work through a simple example. We assume that the files

- main.c
- parser.c
- lexer.c
- makefile

reside in the same directory and that is the directory from which we will invoke *make*.

Consider a projext with the three C files, main.c, parser.c, and lexer.c. These are to be combined into a single executable. So that you can work through this in the exercises, we'll pick these to be essentially stubs of some final large mythical programs.

main.c

```
#include “proj.h”
extern int parser();
int main() {
  printf(“main\”);
  parser():
}
```

parser.c

```
#include “proj.h”
#include “parser.h”
extern int lexer();
int parser() {
  printf(“parser\”);
  lexer():
}
```

lexer.c

```
#include “proj.h”
extern int lexer();
int lexer() {
  printf(“lexer\”);
}
```

To combine these into a single executable, app, we would do the following sequence of steps:

```
gcc -c lexer.c

gcc -c parser.c

gcc -c main.c

gcc -o app main.o parser.o lexer.o
```

As the project moved along we would work on one part or another and periodically recompile the appropriate parts, keeping careful track. The *make* utility automates this and the up front work of writing the *makefile* is not a great burden. In fact, it helps organize your approach to the problem. For our example, we'll start with the *makefile* below.

makefile

```
CC = gcc

app: main.o lexer.o parser.o

  $(CC) -o app main.o parser.o lexer.o

main.o: main.c proj.h

  $(CC) -c main.c

parser.o: parser.c proj.h parser.h

  $(CC) -c parser.c

lexer.o: lexer.c proj.h lexer.c

  $(CC) -c lexer.c
```

Comparing this to the manual steps indicates the flavor of what is happening, but let's go through it and then try some variations.

first line

```
CC = gcc
```

Here CC is a user defined variable, which has been given a value of ‘gcc’, clearly intended to identify the compiler. When it is later used, it will be enclosed to look like this

```
$(CC)
```

If we wanted the capability to compile for subsequent debugging, but wanted to be able to turn that feature off, we could write

```
CC = gcc

# CC += gcc
```

The ‘#’ designates a comment, so to turn on the debugging support, we would just remove the ‘#’.

remainder of the makefile

The rest of the file consists of rules. *Make* will parse each rule and find out

- what it should build - the target
- what the target depends on - the dependencies
- how to build the target - the command(s)

The format is

```
target: dependency list

list of commands
```

Important note: For historical reasons, too deeply entrenched to change, a command must begin with a tab character. Some editors can be configured to substitute the appropriate number of spaces for the tab character. This will not work for *make*.

As an example, consider this rule taken from our *makefile*:

```
app: main.o lexer.o parser.o

  $(CC) -o app main.o parser.o lexer.o
```

Here, *app* is the target. It depends on the files main.o, parser.o, and lexer.o (the dependencies). The second line tells make how to build *app*. Now *make* is smart enough to parse the whole file and figure out that the dependencies form a hierarchy and then knows which commands to do first. Further, *make* will not do unnecessary work. It will not bother to rebuild files whose constituent components have not changed.

a more terse makefile

Here is a *makefile* that works just like the earlier one.

```
app: main.o lexer.o parser.

main.o: main.c proj.h

parser.o: parser.c proj.h parser.h

lexer.o: lexer.c proj.h lexer.c
```

We've taken out all the commands. This will work because *make* can deduce the needed commands.

In this section we've introduced only the tip of the iceberg. We urge you to not only start using make (if you don't already) and get the *GNU make* documentation to learn its many other capabilities. As is typical with other UNIX derived tools, if you wish it had some particular feature, chances are very good that it already does.

1 - 8 Activities/Exercises

The exercises assume you have read through the entire chapter.

1 - 8.1 Logging on

Log on your workstation. Try the command

```
ls -l
```

and then

```
ls -al
```

What does the additional *a* switch do? [see man ls]

Try the command

```
sh -version
```

What does this tell you?

1 - 8.2 Some Standard directories

Using the command ls -F, check out the contents of each of the standard directories from Table 1 of this chapter. For example, try

```
ls -F /dev
```

and so on. What does the *F* switch do?

1 - 8.3 The /proc directory

Look at the contents of the /proc directory. Examine these files in /proc with cat:

- cpuinfo
- meminfo
- interrupts
- ioports
- pci

If any of the cat displays are more than a screen full, pipe the display through more e.g.

```
cat /proc/pci | more
```

1 - 8.4 More about ls

Try ls with the f and t options:

```
ls -f  or  ls -lf
```

```
ls -t  or  ls -lt
```

What do these options do?

1 - 8.5 mkdir and tree

Using *mkdir*, make a directory hierarchy in your home directory e.g.

```
mkdir projects
```

```
mkdir projects/proj0
```

```
mkdir projects/proj1
```

and so on, adding breadth and depth. Then try the *tree* command, if it's present on your system, e.g.

```
tree projects
```

1 - 8.6 tar

Archive and compress your directory hierarchy from the prior problem i.e.

```
tar cvfz proj.tgz projects/
```

Now try to remove the directory e.g.

```
rmdir projects/
```

Why didn't that work? Now try

rm -rf projects/

Was it removed this time? Now restore it via

tar xvfz proj.tgz

In the two instances where we used *tar*, the switches used were *c*, *v*, *f*, *z*, and *x*. What does each mean?

1 - 8.7 Use vi

Create two different multiline files with vi, file1.txt and file2.txt. We'll use them subsequently.

1 - 8.8 The cp and mv commands

Try the *cp* command:

cp file1.txt file3.txt

cp file2.txt file4.txt

Use *ls -l* to verify that the new files were made. Then try

mv file3.txt file5.txt

mv file4.txt file6.txt

Now what does *ls -l* tell you?

1 - 8.9 Using chmod, chown, chgrp

See the man pages for these commands. Make some files and try them out. You'll be less restricted if you have root permission.

1 - 8.10 Using pipes

Try something like we described in Section 5.3 e.g.

cat file1.txt file2.txt | sort > file12.txt

Describe what this command has done.

Now try this

```
ls -l | wc -w
```

What does this do? Does the result make sense? [see man wc]

1 - 8.11 Using gdb

Investigate gdb per Section 1-7.2.

1 - 8.12 Using GNU make

Review Section 1-7.3. With an editor create the files

- main.c
- parser.c
- lexer.c
- makefile (either version)

where all reside in the same directory. You'll also need the corresponding header files, which can be empty. Create this with touch i.e.

```
touch proj.h

touch parser.h

touch lexer.h
```

Carry out these activities:

- Now invoke make and note what is reported to the screen and what new files are created.
- Try removing some *.o files, one at a time, and do another make. Does what make reports to the screen make sense?
- Try removing one of the header files. What happens?

1 - 9 References

Throughout this book we will provide chapter references where appropriate. Here in this chapter our focus is getting up and running on Linux.

There are many books available on Linux today, some are somewhat distribution specific. However, the Linux Documentation Project provides downloadable HOWTO's, Guides, and man Pages that are quite helpful despite the fact that some are a bit out of date. The url is http://www.linuxdoc.org/docs.html.

Here we reference three of the guides:

Paul Sheer, *Rute Users Tutorial and Exposition*, LDP (March 2000). Note: This is a work in progress.

Matt Welsh et al., *Installation and Getting Started Guide*, LDP (March 1998).

Lars Wirzenius and Joanna Oja, *The Linux Systems Administrators' Guide*, LDP (October 1999). Note: This assumes that the reader has already assimilated material such as that in the prior reference.

The Free Software Foundation (GNU) has a large selection of documentation. Here we point out the gcc, gdb, and make manuals (see http://www.fsf.org/).

Richard M. Stallman, Roland Pesch, Stan Shebs, et al., Debugging with GDB, Free Software Foundation, Inc. (May 2000).

Richard M. Stallman, Using GNU CC, Free Software Foundation, Inc. (November 1995). There are more recent on-line versions available at http://www.gnu.org/

Richard M. Stallman and Roland McGrath, GNU Make, Free Software Foundation, Inc. (May 1996).

Another important source of information is the man (for manual) page system which is an online documentation system that comes with the typical Linux distribution. For example, let's say you want to find out about shell scripting with the bash shell. Then you merely enter at the command line:

```
man bash
```

and you'll get a a document about bash displayed on the console.

The typical Linux distribution also comes with documentation in other directories such as:

/usr/doc

/usr/src/linux/Documentation

/usr/X11R6/lib/X11/doc

An embedded Linux must save space so some or all of this documentation may be unavailable on the embedded system itself, but some may be available elsewhere e.g. on a CD that came with the embedded version. You may need to do some detective work.

There are also paper-based and web-based journals. Here we mention two that come in both media and are very good:

The Linux Journal - http://www.linuxjournal.com

Linux Magazine - http://linux-mag.com

Finally, note that web searching for a topic of interest can be quite successful.

CHAPTER 2

Overview of the uCsimm/ uClinux Development Environment

2 - 1 Introduction

This course explores embedded system development on the uCsimm, more formally the “uc68EZ328 Single Inline Microcontroller Module”. The uCsimm as used in this course has uClinux as its resident operating system (OS). This is a stripped down Linux appropriate for embedded systems. Our particular version is for the DragonBallEZ MCU architecture, of the Motorola m68k family.

The course is intended for those who are familiar with the C programming language, the Linux OS, and who have interest in learning about embedded systems development. The course is less appropriate for embedded systems gurus; although, the early portion of the course can serve to introduce the uCsimm/uClinux target. The course is organized as a hands on experience and the participants will:

- learn how to set up and configure the system
- learn what resources are available
- learn to use a selection of the available resources

Embedded system software can be characterized by two extreme cases. In the first case, the software driving the system might be a totally customized, application specific package, with very narrow focus. At the other end of the spectrum we might find the case with an application specific software package that lies on top of

an underlying, somewhat general purpose embedded OS, which provides various capabilities for the application from its general tool set. In the first case, the extra overhead of the general purpose OS might be undesirable, particularly if memory size is extremely limited. On the other hand, the situation with an underlying OS should offer flexibility in design, easier maintenance, and less reinvention of the wheel. With the consistent historical trend being toward more RAM and FLASH ROM in smaller chip real estate (and much cheaper as well) the embedded OS becomes more attractive. However, the embedded OS has traditionally had another drawback - relatively high cost. This raises the per unit cost so that the OS once again becomes undesirable. The OS-resident alternative could prove non competitive in market price or, in the case of a relatively under capitalized startup, out of reach in terms of initial development cost. Hence, the expense of an embedded OS may force a choice based on economics rather than technical merit. This explains the impetus for an embedded version of Linux. In this course we look at a particular example, uClinux. As expected, the source code is open.

Clearly, a product like uClinux fills a need because of its cost advantage. There is, however, another important advantage. Consider a developer, using a system with an embedded OS, who has run up against an intractable bug with the application specific software under construction. It is possible that the bug is in the software under development, but could also be in the OS or based on a misunderstanding of some aspect of the OS. It can be extremely helpful to have access to the actual OS source code and to the community of fellow developers via the appropriate Internet list.

2 - 2 An Overview of the Development Environment

The development environment consists of these elements

- the target hardware and its resident software
- the development workstation
- the interconnection between these two machines

Let’s look at each of these elements, in turn, starting with the target. The target hardware for this course is the uCsimm, a standard 1” tall 30-pin SIMM with various resources. For our discussion at this point, we just mention a few

- an ethernet controller
- a serial port
- a resident OS, uClinux

Further, this course will use an inexpensive companion, the uCgardener. The latter provides a socket for the uCsimm and appropriate peripheral hardware so the developer can get at the uCsimm resources. The uCgardener provides

- the SIMM socket
- an RJ45 ethernet jack (female)
- a DB9 serial connector (female)
- a power jack and on board voltage regulator

The second element of the development environment is the workstation. It is essentially a linux box which should have

- an available serial port
- terminal emulator software (minicom is included in most Linux distributions)
- an ethernet card
- a CD ROM reader

This is, then, a rather typical Linux box. The CD ROM capability is needed to access the uClinux System Builder Kit which comes on a CD. This contains the development tools, called tool chains. Note that it will be typical (but not necessary) that the workstation has an Intel derived CPU. In our case the target will be the Motorola DragonBallEZ MCU, definitely not Intel. Hence, the toolchains comprise a cross compilation tool set.

It is of particular interest that the target OS is Linux, as is that of the development workstation, despite the different CPU's. Of course, the target OS is a very stripped down Linux. The virtue of the open source nature of Linux is apparent - it can be modified appropriately for its host.

The third and last element of the development environment is the interconnection between the workstation and the target. There are potentially two such connections:

- the RS232 serial connection
- the 10 Mbps ethernet connection

Let's consider the serial connection. The workstation, running a terminal emulator (we later assume minicom), becomes the target's terminal. When power is applied (or the uCgardener reset button pressed) the uCsimm boots into the 'Bootloader', essentially a system monitor with a set of commands to which it will respond. Among these commands, available before booting into uClinux, are:

- a command to upload a new uClinux image from the workstation to the target's RAM
- a command to write the RAM image into FLASH ROM
- a command to boot the uClinux image in the FLASH ROM
- an alternative command to boot an image in RAM

At the completion of the boot/initialization of uClinux it presents a login prompt. In summary, the workstation can issue commands to the Bootloader and, once uClinux boots, the workstation becomes the Linux terminal.

During or subsequent to the uClinux boot, the ethernet capability is brought up. The appropriate work area on the workstation can be mounted via nfs, integrating that work area with the uClinux file hierarchy. This becomes really convenient - for example, instead of using the relatively slow serial connection to upload a new uClinux image to the target from the workstation; you can use *flashloader*, a command that has been added to uClinux. This will load an image file into the uCsimm memory, employs the bootloader to load that image into FLASH ROM, and finally warm boots the new OS image.

In comparing the convenience of the ethernet connection to the serial, keep sight of the fact that a usable ethernet connection between target and workstation cannot exist before uClinux boots. If the uClinux image is such that the boot fails, the serial connection will be necessary to save the day - because, under these conditions, only it can be used to transfer a new image. In this respect, it should be recalled that the bootloader not only has a command to boot the FLASH ROM image, but another to boot an image in RAM. These are currently mutually exclusive, but the ability to test code via an nfs mounted partition of your Linux workstation makes testing a boot image in RAM less urgent.

It should also be stressed that the serial and ethernet capabilities of the uCsimm are not just for the development process; they are also available to whatever application is deployed. For example, the embedded device can therefore be Internet ready.

2 - 3 An Overview of the Development Process

The developer uses the Linux workstation for the design and implementation of the embedded application. The usual resources of a Linux workstation are available including the X Window System and familiar editors such as emacs, vi, or nedit and so on.

Once the source code for the application is ready for testing, it is compiled, incorporated into a uClinux image, and transferred to the uCsimm RAM, either by serial upload or via the *flashloader* command which relies on the nfs mounted directory. Once the image is in the uCsimm RAM, it can be run from RAM or loaded into the FLASH ROM and then run. It should be noted that an image built to be run from RAM is different from one to be run from the FLASH ROM; i.e. these images are mutually exclusive. This will be examined more closely in Section 2-6.4 of Chapter 2.

As the development process moves between the Linux workstation OS and the target OS, the developer continues to work at the same keyboard and monitor since it serves both the (possibly Intel) Linux workstation and the DragonBallEZ MCU uCsimm.

2 - 4 Activities / Exercises

The following activities are meant to get the development system to a state of readiness. In the next chapter, we'll get the associated software up and running, after making the serial and ethernet connections.

2 - 4.1 Project Ideas

Using Chapter 2 of the uCsimm/uC68EZ328 Hardware/Software Manual as a guide, describe in some detail at least 3 embedded projects that could be based on the available resources of the uCsimm/uClinux target. Alternatively, read through the many ideas put forward in the uClinux mailing archives. See http://www.uclinux.org/pub/uCsimm/archive/.

2 - 4.2 Minicom

Ensure that minicom and the corresponding man pages are available on your Linux workstation.

CHAPTER 3

Configuring the uCsimm/uClinux

3 - 1 Introduction

Upon getting a system, many of us want to get it up and running immediately, despite not understanding all the relevant details. That is the purpose of this chapter. We will gain a more thorough understanding of the details in the next chapter.

Here we intend to configure and deploy the three elements of our development system. Recall that these elements are

- the Linux workstation
- the uCsimm target
- the interconnection of those machines

  RS232 serial

  ethernet

We'll discuss these elements throughout this chapter and end up with a development environment that is ready for productive work. Note that there are relevant, useful (somewhat interrelated) web sites supporting this setup:

- www.uClinux.org
- www.uClinux.com

- www.lineo.com/cgi-bin/rightnow/

The first gives access to the very useful uCsimm email forum, as well.

3 - 2 Configuring the Linux Workstation

3 - 2.1 Installing the tool chains

To set up your Linux workstation there are perhaps two choices. One assumes that you have an existing Linux box for this work, but you use it for many other activities. In this case you probably have invested significant effort in configuring this box just the way you want it, so that installing the tool chains from the uClinux System Builder Kit CD into the existing setup is the clear choice. On the other hand, if you are dedicating a linux box to this activity and need to do a fresh install anyway, you can do so from that same CD. Describing the Linux installation from scratch is outside the scope of this course and is well documented elsewhere. We note that the fresh install from the uClinux CD results in a system with the desired tool chains already integrated. At any rate, we assume the first case here - that you have a Linux box already up and running with the carefully configured Linux distribution of your choice. Then we need only load the tool chains from the uClinux System Builder Kit CD.

The uClinux/m68k tool chains can be installed as binaries or built from source. We'll only discuss the 'recommended' case, installing the binaries. Your Linux version is likely recent enough to have updated to libc6 shared libraries rather than the older libc5, but we'll describe both possibilities. To see which generation of the shared libraries is on your Linux box, you can type at the command line, 'ldd /bin/ls'. The responses for the two cases will look something like this:

libc6 result

```
libc.so.6 => /lib/libc.so.6 (0x2aac6000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x2aaab000)
```

libc5 result

```
lib.so.5 => /lib/libc.so.5 (0x4000a000)
```

For the libc6 case, become root and

- mount the uClinux CD, e.g. ‘mount -t iso9660 /dev/cdrom /mnt/cdrom’
- change into the libc6 directory, e.g. ‘cd /mnt/cdrom/RPMS/libc6’
- then do a make i.e. ‘make’

The above assumes a CD mount point of */mnt/cdrom*, which you should change as needed to match your system.

The libc5 case is analogous; as root

- mount the uClinux CD e.g. ‘mount -t iso9660 /dev/cdrom /mnt/cdrom’
- change directories like this ‘cd /mnt/cdrom/RPMS/libc5
- then ‘make’

The installation may take some time, depending on your machine characteristics, but when completed you should find some items like these at the top level of the new directory /opt/uClinux:

```
bin/              include/   linux/          man/

deftemplate.sh*   info/      m68-coff/       romdisk/

dev/              lib/       m68-pic-coff/   src/
```

3 - 2.2 Installation Glitches

It is possible that the above not go smoothly, depending on the existing Linux distribution. In particular, the *make* may encounter dependency problems so that you need to replace the *Makefile* on the CD in directory RPMS/libc6 or RPMS/libc5, as appropriate, and remove dependency checking. Specifically, lines beginning *rpm -i* must be changed. As an example,

```
rpm -i uC-src-0.9.2-2.i386.rpm
```

would become

```
rpm -i --force --nodeps /mnt/cdrom/RPMS/libc6/uC-src-0.9.2-2.i386.rpm
```

where we have again assumed a CD mount point of */mnt/cdrom* and that libc6 is appropriate for our machine. Of course, the mount point becomes germane because the new *Makefile* cannot be written onto the CD.

Another possible problem can arise from certain RedHat derived distributions whose *genromfs* utility does not properly populate the */dev/* directory. This can lead to a uClinux boot process which cannot open a console for login and, in effect, hangs. The solution is to remove the existing RedHat *genromfs* and installing that which comes on the uClinux CD.

3 - 2.3 Setting up the working environment

With the tool chains installed, these are the remaining steps to create your working environment for this course:

- Make a directory to hold your work
- Change to that directory
- Then issue the ‘buildenv’ command
- Finally issue the ‘make’ command

For example, the above activity might look like this at the command line:

- mkdir /home/my_username/uCsimm_work
- cd /home/my_username/uCsimm_work
- buildenv
- make

If we then look at the contents of the working directory (/home/my_username/uCsimm in our example), we’ll see something like this:

```
Makefile         image.bin  romdisk/      romdisk.map

deftemplate.sh   linux      romdisk.img   src/
```

Let’s discuss some of these entries.

Makefile

This is placed in our working directory by the *buildenv* command. It is used by *make* to sort out dependencies and build the working directory constituents.

deftemplate.sh

The makefile invokes this default shell script to populate the *romdisk/* directory with the binaries chosen for your project. This default template is a good place to start, but can later be replaced by your own, if so desired. For example, if you created a new template, *mytemplate.sh*, you could use it instead, by using command line syntax of form

```
> make TEMPLATE=mytemplate.sh
```

Note that this script specifies binaries to add to the *romdisk/* directory, but does not remove binaries that are already there. So if there is a binary you want to remove, you must not only remove its reference from the script, but also must remove the binary itself from *romdisk/*.

image.bin

This binary image is the ultimate goal of the make process. It is the image either destined for the uCsimm FLASH ROM or else to be run from the uClinux RAM. Your embedded projects will involve code that will be incorporated into this image. Later we will see how this can be transferred to the uCsimm.

linux

This is a link to the source code hierarchy for uClinux, i.e. */opt/uClinux/linux*. It is of interest to see the m68k specifics. Places of interest are *linux/include/asm* (e.g. check and see what *asm* is linked to) and *linux/arch/m68knommu*.

romdisk/

The uCsimm manual calls this ‘a staging area for the root file system to be built into the ROMfs image’. This directory hierarchy will ultimately be transferred to the uCsimm’s FLASH ROM, so it mustn’t exceed the space available. See the description in deftemplate.sh, above, on how to remove binaries from the image.

src/

This contains source code for the userland binaries and should be a good source of examples for us. You’ll get a chance to explore this in the Section 6.2 of this chapter.

3 - 3 Configuring the uCsimm Target

The uCsimm for this course is snapped in to the uCgardener's socket. The uCgardener does not come pre assembled when purchased, but must be assembled per its accompanying instructions. However, for this course it typically will be pre assembled. In addition, other discrete components have been added to support parallel I/O (via Port D).

The uCgardener needs an external power supply which should be provided at the start of the course or can be obtained locally as indicated by your instructor. The recommendation is a 4.5to 6.0 volt, 200-250 mA AC adapter, with a negative center pin.

Once the serial and ethernet cables described below have connected the uCgardener to the workstation, you can power up the uCsimm target.

3 - 3.1The RS232 Serial Connection

The connecting cable between the Linux workstation and the uCgardener should have a connector at one end to fit the workstation (this typically means the cable at this end needs a female DB9 connector) while the cable end at the uCgardener should be a male DB9. These, generally known as 'straight through 9-pin serial cables' are readily available e.g. from Radio Shack. Your system should already have one of these or your instructor will indicate where to get one in your locality. If the serial cable does not yet connect the uCgardener to the Linux workstation, do so now.

3 - 3.2 The Ethernet Connection

If you are connecting your workstation's ethernet card directly to the uCgardener, the connecting cable should be a Category 5 crossover patch cable with a male RJ45 connector at each end, although it is possible for your Linux workstation to require a different connector. On the other hand, if there is an intervening hub (or switch) the cable should not be a crossover cable. The crossover cables can be slightly harder to find than the 'normal' variety, but are still readily available.

If the ethernet cable does not yet connect the uCgardener to the Linux workstation, do so now. At this point you can apply power to the uCsimm target. In the next section will get the system software up and running.

3 - 4 Getting the Serial Connection Up and Running

When powered up, the uCsimm boots into the bootloader program and sends initial display information and is ready to accept simple commands from its terminal. Of course, the Linux workstation, running minicom or the equivalent, is that terminal. The uCsimm sends its information out the serial port at 9600 bps, 8 data bits, no parity, and one stop bit. For our purposes here, we will assume that you are using minicom.

3 - 4.1 Setting up minicom and the serial connection

To set up minicom on your Linux workstation, invoke it as root by entering *minicom* at the command line. Enter *ctrl-a* followed by *z* (do not hold the ctrl key when you press the z) to get the “Minicom Command Summary” screen. This is configurable and can also be <Alt-z> or <esc><z>. Choose “cOnfigure Minicom” by entering the letter o. This opens a sub menu where you must do the following:

A. Choose the *Serial port setup* and within that choice:

- select your Serial Device (e.g. /dev/ttyS0)
- next set your Bps/Par/Bits as 9600 8N1
- finally say no (N) to both Hardware and Software Flow Control

B. Choose *Modem and dialing* and within that choice

- remove the Init string
- remove the Reset string

C. Choose *Save setup as* .. and save under some name (let’s say uCmin) and when you next start minicom as a user (rather than as root) start by entering *minicom uCmin* at the command line.

You can exit minicom now (from the “Minicom Command Summary” screen). Then you might check the permissions of the serial port device chosen to assure that you have access as a user.

Now restart *minicom* as a user e.g. *minicom uCmin* and press the reset button on the uCgardener. You should see a display something like this:

```
uCbootstrap v1.2 (c) Copyright 2000 Rt-Control All Rights Reserved

FLASH type 2249 [AM29LV160B]

DP|004000 DP|006000 DP|008000 DP|010000 DP|020000 DP|030000

D-|040000 D-|050000 D-|060000 D-|070000 D-|080000 d-|090000
```

and so on, then followed by a *B$* prompt. At the prompt, enter *help* to see a list of commands to which the uCsimm bootloader program responds. If this all works as indicated, the serial connection is established and the uCsimm has successfully booted the bootloader program. Chapter 3 will describe the bootloader program in more detail.

3 - 4.2 Booting uClinux

From the bootloader's B$ prompt, one of the commands available is *go*, which will execute the OS image in the FLASH ROM. As shipped, the uCsimm has a version of uClinux which will boot up.

Try the *go* command and see if uClinux boots and ultimately presents a login screen. Choose a login name (the system as shipped doesn't care) and enter as the password, *uClinux*. If successful at logging in, try some Linux commands e.g. *ls*. If this all works, the uCsimm is operational with its uClinux OS.

Note: If the system has been used since being received, it is possible that the OS image will fail to boot. In that case, a new image must be uploaded from the Linux workstation as described in Section 4.3.

3 - 4.3 Uploading a new OS image

Although it is more efficient to transfer an image via ethernet, the serial method may be necessary in cases where the ethernet connection is not available, For example, one situation you want to avoid is having loaded an unbootable image into FLASH ROM - but bad things happen. In such a situation the serial upload is your only recourse. One can upload a new image to the uCsimm's RAM at 9600 bps(slow), but can also do it at 115,200 bps (fast). We'll describe each case.

Try each with the instructor's guidance:

slow

- At the *B$* prompt type *rx*,

  B$ rx
- Press *ctrl-a* and then *z* to enter the "Minicom Command Summary" screen
- Choose s for *Send files*, then as prompted choose the *xmodem* for upload protocol
- Next you'll be prompted for the filename so choose your directory containing the file, press *CR*, and enter the file name as prompted. The screen will show the progress of the file transfer and prompt you when finished.

Note: It is likely that these steps will take too long the first try, the uCsimm's bootloader will get impatient, and the transfer will fail. Just try again.

fast

- At the *B$* prompt, type *fast*,

  B$ fast
- Press *ctrl-a* and then *z* to enter the "Minicom Command Summary" screen
- Choose p for comm Parameters and change to 115200 bps
- Press CR upon returning from the "Minicom Command Summary" screen
- At the B$ prompt, type rx

  B$ rx
- Press *ctrl-a* and then *z* to enter the "Minicom Command Summary" screen
- Choose s for *Send files*, then as prompted choose the *xmodem* for upload protocol
- Next you'll be prompted for the filename so choose your directory containing the file, press *CR*, and enter the file name as prompted. The screen will show the progress of the file transfer and prompt you when finished.

To move the just uploaded image to FLASH ROM, use the *program* command:

B$ program

which will write the image in RAM to the FLASH ROM. It first erases an appropriate area starting at 0x10c10000. Once that has completed you can run the FLASH ROM image via the bootloader command *go*.

```
B$ go
```

Give this a try with a new image uploaded into RAM.

One can build an alternative image which can be uploaded to the uCsimm's RAM, and then executed from RAM with the *goram* bootloader command. This will be investigated in Section 6.4. of this chapter.

3 - 5 Getting the Ethernet Connection Up and Running

The uClinux boot process, toward the very end, executes the shell script /etc/rc. You can look at the default script in your work area at *romdisk/etc/rc*. One thing the script does is to set up the uCsimm's network parameters as follows:

- IP address = 192.168.1.200
- network mask = 255.255.255.0
- network address = 192.168.1.0
- gateway address = 192.168.1.100
- interface type = eth0

You may decide to change these. For example, if your system is isolated from the external Internet, the gateway is not needed - or you may have a different network address already set up etc.

Later the shell script attempts to nfs mount a directory area on the Linux workstation:

```
/bin/mount -t nfs 192.168.1.11:/home/jeff/kit /usr
```

Clearly, this assumes that

- the Linux workstation IP is 192.168.1.11
- your Linux workstation's working directory is /home/jeff/kit

Neither of these assumptions is likely to be true; so the nfs mount will fail, slowing the boot process somewhat, as well.

The preceding, although probably leading to a failed nfs mount attempt, provides an appropriate starting template. For starters, your Linux workstation

- must be configured for nfs
- must have its network parameters set up consistently with those assumed by the romdisk/etc/rc script discussed above (note that you can change the script choices as well - the two machine just need to agree)

We'll discuss these two items in the next two subsections.

3 - 5.1 Configuring nfs on the Linux Workstation

If your Linux Workstation does not have nfs set up you will need to reconfigure and recompile your Linux workstation to include nfs support. This is outside the scope of this book, but your instructor can help you with this exercise. However, one easily overlooked (hidden?) choice to make during kernel reconfiguration is to respond to the very first choice (Code maturity level options) by choosing "Prompt for development and/or incomplete code/drivers". Otherwise you will not be asked later during configuration about nfs server support, which the Linux workstation will need.

Once nfs is configured you must add a line to /etc/exports to tell the nfs server what files it may export for external mounting. For example, in the uClinux /etc/rc script we had the line

```
/bin/mount -t nfs 192.168.1.11:/home/jeff/kit /usr
```

so the line to add to /etc/exports could be

```
/home/jeff/kit (ro)
```

or to give more flexibility

```
/home/jeff (ro)
```

since it allows anything below that hierarchy node to then be externally mounted. Again, modify the file and directory names to match your system. If further

directory trees need to be exported just add further lines to /etc/exports. See *man 5 exports* for further details.

3 - 5.2 Setting up the network parameters on your Linux workstation

This activity is also outside the scope of this book and is well covered elsewhere. Typically, your Linux startup activities will configure your network parameters. However, we will examine a particularly simple case here where

- Your Linux workstation is not connected to the external Internet
- but is only connected to the uCsimm target by ethernet

This case is essentially that of the isolated and dedicated embedded development environment. Here you can use the *ifconfig* and route commands to configure your ethernet interface. As root, you can execute a script containing something like this:

```
ifconfig lo 127.0.0.1

route add -net 127.0.0.0

ifconfig eth0 192.168.1.11 netmask 255.255.255.0 broadcast 192.168.1.255

route add -net 192.168.1.0
```

Subsequently, as root you can issue an *ifconfig* command with no arguments to see the result of your ethernet configuration attempt.

You must also put an entry in your workstation's /etc/hosts file identifying the uCsimm's IP address. An example of /etc/hosts for our assumed IP addresses would be:

```
127.0.0.1       localhost               lo

192.168.1.11    Linux_Workstation   uCdevelop

192.168.1.100   uCsimm_Target       uCtarget
```

3 - 5.3 Transferring uClinux images aided by nfs

Once the uClinux on the uCsimm boots and successfully nfs mounts your working directory, you have a second method for transferring a newly made image from the

workstation to the uCsimm. Let's say your new image is on your Linux workstation at */home/jeff/kit/image.bin*. Further, assume that */home/jeff/kit* was successfully mounted on the uClinux directory */usr*. The uClinux provides some extra commands among which is *flashloader*, which loads the image file specified into memory and the hands it to the bootloader which then writes it into the FLASH ROM's area designated for the OS - and then warm boots the new image. With our assumption above this would be accomplished on the uCsimm/uClinux with the command:

```
flashloader /usr/image.bin
```

3 - 6 Activities / Exercises

3 - 6.1 Install the Tool Chains and a Working Directory

Refer to Section 3 - 2. However, we will give the sequence of steps here to make it easier to keep track. It is important not to miss a step, so do something like a check off system. All these steps are carried out on your Linux workstation.

Step 1: Check the shared library dependence of your system, i.e. *libc5* or *libc6*, with

```
ldd /bin/ls
```

Step 2: Mount the uClinux CD, e.g.

```
mount -r -t iso9660 /dev/cdrom /mnt/cdrom
```

Step 3: Change to the CD's libc5 or libc6 directory in accordance with the result from Step 1, e.g.

```
cd /mnt/cdrom/RPMS/libc6
```

Step 4: Do a *make*

Step 5: Check that the */opt/uClinux* directory has been created and properly populated

Step 6: Create a working directory in your home directory

Step 7: Change to that working directory and

enter *buildenv*

then enter *make*

Step 8: Check that your working directory has been properly populated as a result of the prior two steps

3 - 6.2 image.bin

Look at the Makefile in the uCsimm working directory.

What are the 2 constituents of image.bin by name?

Explain the content of each of those constituents. How can the content of each be modified?

3 - 6.3 Examining the src/ subdirectory in the uCsimm working directory

Start with the README.

Find the program *login.c*. What username is required to login? What password?

Find the program *shutdown.c*. What signals are sent upon shutdown? What do these signals mean (see the man page, perhaps ‘man 7 signal’)?

Feel free to spend some time rummaging around in this subdirectory.

3 - 6.4 What steps are needed to recompile the uClinux kernel for producing a modified linux.bin?

Note: This is much like a normal kernel recompile where the usual source tree location /usr/src/linux gets replaced by /opt/uClinux/linux.

3 - 6.5 Get the serial connection up and running

Refer to Section 3 - 4. However, we will give the sequence of steps here, to make it easier to keep track. It is important not to miss a step, so do something like a check off system.

Step 1: Using a ‘straight through’ (not null modem) cable, connect your workstation to your uCgardener. Ensure that your uCsimm is properly inserted. Connect the power adaptor to the uCgardener.

Step 2: On the Linux workstation, ensure that xmodem is present with the proper soft links i.e.

/usr/bin/rx --> /usr/bin/rz

/usr/bin/sx --> /usr/bin/sz

Step 3: Configure Minicom (usually using /dev/ttyS0 or /dev/ttyS1) on the Linux workstation.

Step 4: Transfer the original image from the workstation to the uCsimm via Minicom.

Step 5: On the uCsimm (via Minicom), enter ‘program’ to burn the newly transferred image into FLAH ROM. Note: For some reason, this is the step people tend to forget ... and if one then goes ahead to Step 6, the image transferred in Step 4 is obliterated.

Step 6: On the uCsimm (via Minicom), enter ‘go’ to boot uClinux.

3 - 6.6 Create an image appropriate for establishing the subsequent ethernet connection

Choose a network for your Linux workstation and uCsimm target to share. Choose IP addresses for each machine. Note that the ‘as shipped’ default uCsimm expects:

- network address = 192.168.1.0
- uCsimm IP = 192.168.1.200
- Linux workstation IP = 192.168.1.11

You are not required to make these same choices, but assure that all three choices share the same first 3 numbers of the dotted quad notation (e.g. 192.168.1, as above). If you are unfamiliar with the IP address classes, see most any book on network administration e.g. Olaf Kirch's *The Linux Network Administration Guide*, available from http://www.linuxdoc.org/guides.html or in hardcopy from O'Reilly.

Once you have made your address choices:

- go to your working directory on the Linux workstation and edit the file romdisk/etc/rc to match those choices.
- Next from within the working directory, do a make.
- Finally, transfer your new image, following the steps given in exercise 3-6.5.

3 - 6.7 Get the ethernet connection up and running

Refer to Section 3 - 5. However, we will give the sequence of steps here, to make it easier to keep track. It is important not to miss a step, so do something like a check off system. Not: It is assumed that you have already transferred the appropriate uCsimm image in the prior exercise.

Step 1: Using a crossover ethernet cable (not a 'straight through'), connect your Linux workstation to the uCgardener.

Step 2: On your Linux workstation, edit */etc/exports* per Section 3-5.1, so that *nfs* can export your working directory.

Step 3: On the Linux workstation, adjust the IP address of the Linux workstation to match the choices you made in exercise 3-6.6. You can do this per Section 3-5.2.

Step 4: Stop and start the *nfs* server to make it aware of the new addition to */etc/exports*. There have been reports that 'restart' doesn't work properly on some versions, so you might be cautious and use 'stop' and 'start' e.g.

```
/etc/rc.d/init.d/nfs stop

/etc/rc.d/init.d/nfs start
```

Step 5: On your Linux workstation, edit */etc/hosts* so it contains the IP addresses of the Linux workstation and of the uCsimm.

Step 6: Reboot the uClinux on the uCsimm. Perform the following tests:

- on the uCsimm/uClinux target, ping the uClinux IP and then ping the Linux workstation IP, e.g.

  ping 192.168.1.200

  ping 192.168.1.11
- on the Linux workstation, ping the Linux workstation IP and then ping the uClinux IP, e.g.

  ping 192.168.1.11

  ping 192.168.1.200
- on the uCsimm/uClinux target, see if the workstation's working directory was successfully mounted, i.e.

  ls /usr

If all this works, your ethernet connection and *nfs* mounting are in good shape. If not, troubleshoot with this chapter as your guide and with your instructor's help.

3 - 6.8 Try out the *flashloader* command per Section 5.3

CHAPTER 4

The uCsimm/uClinux Resources

4 - 1 Introduction

In this chapter, we'll list the features of the uCsimm/uClinux target. Keep in mind that for our course the uCgardener is used as an inexpensive installation board for the uCsimm that allows us to access the uCsimm resources. Also note that the presence of uClinux provides significant support; e.g., the ethernet hardware becomes TCP/IP capable and the system is Internet ready at the start.

The uCsimm is based upon the uC68EZ328 architecture, which is organized in 3 functional blocks:

- MCU Core
- System Memory
- Ethernet Controller

As already mentioned, in addition to the integrated hardware resources, the system also has software resources, namely,

- the Bootloader
- uClinux

4 - 2 Hardware Resources

Note these important references:

uCsimm/uC68EZ328 Hardware/Software Manual by D. Jeff Dionne and Michael Durant [packaged with the uCsimm].

DragonBall EZ MC68EZ328 User's Manual [a pdf version is available on the uClinux System Builder Kit CD]

For subsequent reference we give the uCsimm pinout next, noting that many of the pins are multiplexed among two and even three functions. The 21 general purpose I/O pins are identified specifically in the 4th column.

TABLE 4 - 4. uCsimm Pinout

Pin #	Function 1	Function 2	General Purpose I/O
1	Ethernet ERX-		
2	Ethernet ERX+		
3	Ethernet ETX-		
4	Ethernet ETX+		
5	PWM output	CTS (RS232)	PB6
6	Timer I/O	RTS (RS232)	PB7
7	VDD (power)		
8	Reset		
9	GND (ground)		
10	STXD (SPI)		PE0
11	SRXD (SPI)		PE1
12	RSRXD (RS232)		
13	RSTXD (RS232)		
14	SCLK (SPI)		PE2
15	/IRQ6		PD7
16	/IRQ3		PD6
17	/IRQ2		PD5

TABLE 4 - 4. uCsimm Pinout

Pin #	Function 1	Function 2	General Purpose I/O
18	/IRQ1		PD4
19	/INT3		PD3
20	/INT2		PD2
21	/INT1		PD1
22	/INT0		PD0
23	LACD (LCD)		PC7
24	LCLK (LCD)		PC6
25	LLP (LCD)		PC5
26	LFLM (LCD)		PC4
27	LD3 (LCD)		PC3
28	LD2 (LCD)		PC2
29	LD1 (LCD)		PC1
30	LD0 (LCD)		PC0

4 - 2.1 MCU Core Block

The static 68EC000 core processor is identical to the MC68EC000 microprocessor and features full compatibility with the MC68000 as well. Running at 16.58MHz it performs 2.7 MIPS. It also provides a UART, SPI, LCD controller, Timer/PWM, and parallel I/O.

The UART with integrated RS232 line drivers provides a single 3 wire port (RSRXD, RSTXD, GND) which can run up to 115200 bps. The system provides a boot strap mode function which allows system initialization as well as program/data download via the UART. If hardware handshaking is necessary, RTS and CTS are provided. However, those pins can be used by the PWM and Hardware Timer or as general purpose I/O (PB6 and PB7).

The 16-bit programmable Serial Peripheral Interface is a standard 3 wire Motorola SPI supporting external peripherals and operating in Master mode. It can run up to 4Mbps and can support a wide range of external peripherals without additional components, including

- UARTs

- DSPs
- SPI slaves

The SPI can also be connected to D/A and A/D converters.

A no glue connection to a single panel monochrome LCD is available with a software programmable screen size up to 640x512. It will support up to 4 levels of grey out of 16 palettes. The LCD driver utilizes system DRAM as display memory. LCD contrast control can be achieved using 8-bit PWM. Touch screen capability can be provided with a small amount of additional external circuitry.

The PWM/Timer provides

- Pulse Width Modulation with 8 bit resolution and a 5 byte FIFO. With external filtering and a transducer driver, it is capable of telephone quality sound generation. It can also be used to provide low accuracy D/A conversion functions such as LCD contrast, as mentioned earlier.
- A 16-bit general purpose counter/timer with 60-ns resolution (at 16.58MHz system clock) and automatic interrupt generation. It includes an input/output pin.

Recall that the PWM/Timer shares pins with the serial RTS and CTS signals. It should be noted that if RTS/CTS is to be used an external RS232 line driver is required.

The uCsimm provides up to 21 parallel I/O pins out of the up to 47 supported by the MC68EZ328. These are in all cases multiplexed with other peripheral functions as can be seen from Table 1. In particular, note that

- PB6 is multiplexed with the PWM output and the RS232 port CTS.
- PB7 is multiplexed with the Timer I/O and the RS232 port RTS.
- Port C (PC0 -> PC7) is multiplexed with the LCD pins.
- Port D (PD0 -> PD7) can be configured as input and optionally generate interrupts. Alternatively, it can be configured as output.
- PE0, PE1, and PE2 are respectively multiplexed with the SPI's STXD, SRXD, and SCLK.

See the uCsimm manual for behavior on reset for the various general purpose I/O. Our uCgardener has added circuitry to support parallel I/O for using Port D as output (leds) and input (switches). This is explored in the next chapter.

4 - 2.2 System Memory Block

The system memory consists of

- 2 Mb Flash ROM
- 8 Mb DRAM

These are configured as 16 bit wide, zero wait state. When idle, a low power shutdown state is automatically entered. Recall that if you are using the LCD controller, it uses some of the DRAM. Although this DRAM sharing is transparent to the user, there are then bandwidth issues. The MC68EZ328 User's Manual shows how to make pertinent bandwidth budget calculations. The system memory map appears in Table 2.

TABLE 4 - 5. System Memory Map

Inclusive address range	Function	Size
0x00000000 - 0x0001FFFF	Bootloader System RAM	128 kb
0x00020000 - 0x007EFFFF	Operating System RAM	8000 kb
0x007F0000 - 0x007FFFFF	Bootloader Stack RAM	64 kb
0x10000300 - 0x10000310	Ethernet Controller	16 b
0x10C00000 - 0x10C0FFFF	Bootloader FLASH Image	64 kb
0x10C10000 - 0x10DFFFFF	Operating System FLASH	1984 kb
0xFFFFF000 - 0xFFFFFDFF	DragonBallEZ Registers	3.5 kb
0xFFFFFE00 - 0xFFFFFFFF	DragonBallEZ Boot Microcode	0.5 kb

I/O Memory

As shown in the preceding Table, the ethernet controller is mapped as 16 bytes beginning at 0x10000300. Our access will be via the driver packaged with uClinux, so we will not need to program this device directly.

The memory area from 0xFFFFF000 through 0xFFFFFDFF consists of various MC68EZ328 registers. This space includes I/O related registers as well, such as the direction and data registers for ports A through G. Earlier, in Table 1 of this chapter, we saw what subset of the Dragonball EZ I/O ports are actually available on the uCsimm. The interested reader is referred to Table 1-3 in the Dragonball EZ User's Manual for the detailed memory map for this region. In our programming, we'll refer to the I/O ports by name and will not need to know their corresponding

numerical addresses. Nevertheless, the aforementioned table is worth perusing just to see what it includes. Recall that the manual is included on the uClinux CD.

FLASH ROM

The FLASH ROM is a 29LV or 29DL series, 3.3 Volt device. As we'll see later, the bootloader has commands for managing the FLASH ROM. Of course, using these commands is preferable to the less prudent alternative of programming the hardware directly.

DRAM

The uCsimm provides 8 Mb of EDO DRAM. The DragonballEZ MCU handles the refresh of the DRAM even when the MCU is in its low power sleep mode.

After reset, the bootstrap runtime code configures the DRAM and copies the bootloader code and relevant data to the first 128 kb of DRAM. This includes environment variables, default fault handlers, and debug stubs. Hence, care must be taken not to write into the first 128 kb of DRAM.

4 - 2.3 Ethernet Controller Block

The ethernet controller is a CrystalLan CS8900A with all necessary support circuitry to provide a complete 10BaseT ethernet port. Our uCgardener provides an RJ45 female connector. With uClinux providing the driver code, it is ready to run (see Chapter 5). When the network capability is not required, the ethernet controller enters a low power sleep mode.

4 - 3 Software Resources

4 - 3.1 The Bootloader

We have already made use of the bootloader software, but it has other capabilities which we'll catalog here.

uCbootstrap

As shown earlier in Table 2, the bootloader resides in a 64 kb portion of the FLASH ROM. It initializes the hardware to a known state, tests the hardware, and then (in

its default behavior) presents the user with the bootloader shell. We'll examine the bootloader command set shortly.

uCbootstrap also hooks the TRAP #2 vector to provide system calls so that it is possible to make bootloader system calls from your C code - not needed by typical applications. This is an advanced topic and outside the scope of this course. If you desire to pursue this topic, note that there is some information in the uCsimm manual and that you will need to familiarize yourself with the uClinux include files to be found in src/boottools/include. However, the files contained therein have been truncated to discourage the casual use of such bootloader system calls. Nevertheless, the complete files are available by request and from the mailing list.

We note that the associated API provides various capabilities including those to

- reset the module
- manage the FLASH ROM
- read/write environment variables

Finally, the uCbootstrap hooks exception vectors and tries to recover from unexpected traps.

Bootloader Commands

If you enter *help* at the bootloaders *B$* prompt, you get a list displaying each command with a terse description. Next we'll tabulate them in that same order, giving a more thorough description of each. Optional parameters appear in square brackets.

TABLE 4 - 6. The bootloader commands

command	description
sh	Recursively invoke a new bootloader shell.
exit	Exit from the current shell. If the shell exists because it was invoked from an exception handler, the bootloader tries to return from the exception and continue execution. If there is nothing to return to, a new shell will be spawned.
help	Print a list of commands with short descriptions.

TABLE 4 - 6. The bootloader commands

command	description
printenv [name]	Display environment variables as allowed by the current environment variable protection mask. When used without the optional argument, all environment variables are displayed, if allowed by the protection mask. If the optional argument is used, *name* specifies a single environment variable for display, if permitted by the protection mask.
setenv name [value]	Set the environment variable specified by *name* to *value*. If *value* is not specified, the environment variable *name* is erased. Creating new environment variables or erasing existing environment variables is done in a manner consistent with the current protection mask.
eraseenv	Erase the FLASH block containing environment variables which can be modified by the user. This command disregards permissions.
pmask [bsu+-rw]	Set or display the current environment variable protection mask. With no arguments, the current protection mask is displayed. With arguments, the mask is set as specified by those arguments. For example, pmask u-w would remove write permission from the user domain. The domains are *b* (bootloader), *u* (user), and *s* (supervisor).
rx	Receive a binary image via the RS232 port using the XMODEM protocol. Store the new image starting at DRAM address 0x00020000, right after the bootloader RAM area.
program	Erase an area of the FLASH ROM starting at 0x10c10000, right after the bootloader FLASH image. Then write the image currently in DRAM into that area of the FLASH. The image is presumably one that was received into DRAM via the *rx* command above.
verify	Verify that the image in DRAM matches that in the OS area of the FLASH ROM.
go	Execute the image residing in the OS area of the FLASH ROM. In our typical case, this will result in starting up our current incarnation of uClinux.

TABLE 4 - 6. The bootloader commands

command	description
goram	This executes the image residing in DRAM, presumably one that has been received via the *rx* command. Currently this command requires a patch and doesn't coexist gracefully with the prior *go* command.
fast	Change the serial speed to 115200 bps. Once this is done you must also change the speed of the terminal emulation program (e.g. *minicom*) on the Linux workstation.
md address [endaddress]	Display a memory dump in hexadecimal, starting at *address* and ending at *endaddress*. If *endaddress* is not specified, exactly 16 bytes will be displayed.
mm address values...	Starting at *address*, sequentially write a byte at a time from *values* into memory. For example, *mm 00020000 123456* will write bytes as follows: 0x12 into 0x00020000 0x34 into 0x00020001 0x56 into 0x00010002 Note: Since this writes to memory, it is possible to do so improperly and cause problems as severe as a module crash.
envmm	Read the environment variable pairs of form *>address values...* and write a byte at a time from *values* into consecutive memory locations starting at address. For example, *setenv >00020000 1234* *setenv >00020002 56* *envmm* will write bytes as follows: 0x12 into 0x00020000 0x34 into 0x00020001 0x56 into 0x00020002 Note: Since this writes to memory, it is possible to do so improperly and cause problems as severe as a module crash.

Special Environment Variables

In this subsection we'll examine special environment variables which can affect module operation or its bootup sequence. these appear in Table 4. The last three entries in the table carry cautionary notes.

TABLE 4 - 7.

Environment Variable	Description
FACTORY	The copyright string for the uCsimm design
REVISION	The revision number of this uCsimm
HWADDR0	The hardware address of the single network interface
SERIAL	This module's serial number
CONSOLE	If this is specified as yes or ttyS0, the serial port is initialized to 9600, 8, n, 1 and is used as the console. Otherwise, no console is configured.
AUTOBOOT	If this variable is a number, the bootloader will execute the OS image in FLASH after that number of seconds has elapsed, unless a character is entered at the console before that time. If *AUTOBOOT* is *yes* and *AUTOKEY* is properly set, the module boots into the OS with no delay. Any other value for *AUTOBOOT* will generate an error message. Note: There is a dangerous possibility here. If *AUTOBOOT* and *AUTOKEY* are set for immediate OS image execution and the OS image is faulty, there is no way to replace the OS image in FLASH.
AUTOKEY	If set to *iknowmyimageworks* with *AUTOBOOT* set to *yes*, the module executes the OS FLASH image directly, with no possibility of intervention via the console. Note: There is a dangerous possibility here. If *AUTOBOOT* and *AUTOKEY* are set for immediate OS image execution and the OS image is faulty, there is no way to replace the OS image in FLASH.
ENVMM	If set to *auto*, the *envmm* command is run upon boot. Note: Since this writes to memory, it is possible to do so improperly if one of the >environment variables has an inappropriate value and cause problems as severe as a module crash.

An Example: LCD display refresh at bootup

Recall the last entry in Table 4. If *ENVMM* is set to *auto*, the bootloader sees this at bootup and executes the *envmm* command. Our example here uses that capability so that the LCD display will be refreshed at bootup. In particular, the uCsimm LCD controller is assumed to be connected to the EPSON EG9013 Mono VGA LCD panel.

This involves setting up appropriate LCD related registers as well as two related to port C. The details aren't particularly important, but the scenario is

- use setenv to set up appropriate environment variables
- use envmm to actually write those values to memory thereby testing the result
- then only if the display is OK, set ENVMM to auto

A subsequent bootup will have the panel display from a frame buffer at 0x00020400. From the uCsimm manual, the detailed steps are as follows:

From the bootloader command line enter these commands (comments in braces):

```
setenv >fffffa00 00020400   {LCD 32 bit screen starting address}

setenv >fffffa05 28         {LCD 8 bit virtual page width}

setenv >fffffa08 0280       {LCD 16 bit screen width}

setenv >fffffa0a o1df       {LCD 16 bit screen height}

setenv >fffffa29 00         {LCD 8 bit LCD refresh rate adjustment}

setenv >fffffa25 00         {LCD 8 bit pixel clock divider}

setenv >fffffa20 08         {LCD 8 bit panel interface configuration}

setenv >fffffa21 01         {LCD 8 bit polarity configuration}

setenv >fffffa27 81         {LCD 8 bit clocking control}

setenv >fffff412 ff00       {Port C pull-down enable and select}

envmm                       {try out the prior settings}
```

If all looks OK, then enter

setenv ENVMM auto

A different panel would likely have different specifications so the preceding values would need to be changed appropriately. For first attempts at using a LCD panel, it would be wise to check the mailing list to find one that seems to function properly with the uCsimm.

4 - 3.2 uClinux

One of the virtues of Linux is that its open source nature allows it to be tailored to various targets i.e. not only different CPU's, but also a different mix of other hardware resources. For example, X Windows is quite a large piece of software, but rides on top of the kernel and can be abandoned for an embedded target. Alternatively, it could be replaced by a simpler GUI interface. The constraint for the uCsimm is that whatever is chosen to be included must lead to a uClinux that fits into the FLASH ROM area reserved for the OS (see Table 2, earlier in this chapter).

uClinux arose from an effort to port Linux to micro controllers without memory management units. It is currently patched from the 2.0.38 Linux kernel. However, uClinux is a much smaller derivative with:

- uCkernel < 512 kb
- uCkernel plus tools < 900kb

It retains the familiar Linux API with network and file system support (e.g. NFS, ext2).

To see what is in the uClinux root file system, examine the romdisk directory in your working area. You might also examine deftemplate.sh to see which Unix utilities have been commented out and might be included.

Booting uClinux

When Linux boots it executes /sbin/init which in turn executes the shell script */etc/rc*. The latter does various tasks including

- setting up the hostname
- attaching the network interface

- expanding the ramdisk
- mounting /dev/ram0, mounting proc, and mounting nfs
- starting up the internet daemon

The program /sbin/init next starts up whatever is listed in */etc/inittab.* As initially configured, *inittab* sets up the serial line for terminal use and spawns */sbin/agetty* to manage the serial console which is actually your Linux development station running a terminal emulation program such as minicom.

As mentioned in an earlier chapter, you may log in with any username (as long as there is something before the <cr>), while the expected password is *uClinux*. This logs you in as user; you cannot log in as super user. Your ability to affect the kernel is in building *image.bin*. You may be accustomed to having root privileges if desired at login - appropriate for your own workstation, but perhaps not for an embedded system.

Extra/Replacement Commands

uClinux contains a subset of the commands found in Linux on your workstation, but also includes some additional or replacement commands. If one of these special uClinux command names is the same as one existing in Linux, the former is a replacement. Here is a list of these commands, each with a short description:

- expand

  usage: expand infile outfile

  Expands an infile into outfile. It is used during bootup in */etc/rc*.

- flashloader

  usage: flashloader FLASH imagefile

  Loads an imagefile (e.g.image.bin) into uClinux DRAM and hands control to the bootloader which writes the imagefile to the OS area of FLASH ROM and then executes the image file in FLASH ROM, thereby booting the new OS.

- httpd

  usage: httpd [-i]

  This is a WWW server. Without the optional parameter, it uses a hard coded document root, normally */htdocs*. With the -i parameter, it reads the request from standard input and sends the result to standard output (for use with inetd).

- ifattach

  usage: ifattach [--addr x.x.x.x] [--mask x.x.x.x] [--net x.x.x.x]

  [--gw x.x.x.x] [iface]

  This configures the interface as given by the parameter list and then sets up routing per the netmask and network address. If --gw address is given, the default route is set to go through the gateway specified. Unspecified parameters have these effects:

  no --addr, defaults to 127.0.0.1

  no --mask, defaults to 255.0.0.0

  no --net, defaults to 127.0.0.0

  no --gw, no default route

  no iface, defaults to lo

  See */etc/rc* for usage examples both without and then with parameters.

- inetd

usage: inetd

This daemon reads */etc/inetd.conf* for the ports it must listen on for incoming connection requests. When a request comes in, the network connection hooks up to the standard input, output, and error of the specified program, spawned to deal with the connection. An example line from */etc/inetd.conf* is:

telnet stream tcp nowait root /sbin/telnetd

which indicates that an incoming connection on the telnet port will spawn */sbin/telnetd*. To find out more about the identity of the telnet port, inetd looks at */etc/services* and finds that the telnet port is 23.

- init

  usage: kernel automatically runs this at boot

  This starts up the system and is the parent of all processes. It runs the shell script */etc/rc*. Subsequently, init keeps the processes listed in */etc/inittab* running. As mentioned earlier, in its as shipped configuration, this involves only */sbin/agetty* which is the console task.

- login

  usage: login [-t]

  This is typically run from agetty or telnetd when a user attempts login. Without the optional parameter, it asks immediately for a password. With the -t option it first asks for the username.

- reset

  usage: reset

  This resets the system, rebooting the bootloader, but not uClinux (unless set to autoboot).

4 - 4 Activities / Exercises

4 - 4.1 Feature Overlap

Review Table 1 and delineate all pins which can be used for multiple features e.g. what are the possible uses of pin 5?

4 - 4.2 bootloader commands

By now we've actually used some of these commands e.g. program and go. Try a few more as follows:

pmask - what is the current mask? Try changing then restoring that combination.

md - look at the memory map and find out where the FLASH ROM area holding the uClinux image is and display the first 16 hexadecimal bytes. Confirm this by looking at the appropriate image file on your workstation with a hex editor e.g. *hexedit image.bin*

printenv - display the environment variables in Table 4. Are there some that don't show? If so, change one in a safe manner so it displays - then restore it as it was.

verify - Try the verify command in two situations; one where verification should succeed and one where it should fail. How does verify communicate success and failure?

4 - 4.3 uClinux Extra/Replacement commands - inetd

We mentioned that the uClinux inetd daemon reads /etc/inetd.conf for ports to watch for incoming connection requests. Check that file to see if incoming telnet and http requests are to be honored. If so, attempt to access the target from your Linux workstation:

- with a www browser for http access
- with telnet for remote login

What root file is the browser accessing?

4 - 4.4 uClinux Extra/Replacement commands - reset

Try the *reset* command at the uClinux command line. Is it implemented? If not read through *deftemplate.sh* in your working directory. Then figure out what to do to get a uClinux image that supports reset. *What* other command might you implement in the same fashion. Now do so; i.e., get a new image into the FLASH ROM that supports these commands.

CHAPTER 5

Using the General Purpose Parallel I/O

5 - 1 Introduction

The uCsimm has available 21 general purpose I/O points. These in many cases are multiplexed with other capabilities (see Table 1 in Chapter 4). For our course, circuitry has been added to the uCgardener to make Port D available as input or output. In particular, there are now available 8 output leds and 8 input switches. The circuitry uses negative logic:

- bit == 1 means off
- bit == 0 means on

An implementation in positive logic would have involved one more part.

Note: When using an I/O point as output, the input switch should be in the off position. If not sure which position is off, check with your instructor. With these 8 I/O points from Port D, we can introduce examples of writing and reading such general purpose I/O points. The remainder of this chapter works with Port D, starting with three example programs in the next section.

5 - 2 Two Example Programs using Port D I/O

In this section we look at two example programs:

- Program #1 - output only, turns on leds
- Program #2 - input only, reads switches

5 - 2.1 Example Program #1 - Write Port D

Our first program asks the user which led to turn on and then does so.

A listing of write_led.c follows:

```
/*
  filename: write_led.c
  programmer: R.A. Sevenich
*/

#include <asm/MC68EZ328.h>

void init_portd(void)
{
   /* select port D bits for i/o */
   PDSEL = 0xFF;
   /* set bits 0-7 as output */
   PDDIR = 0xFF;
   /* Initialize to off */
```

```
  PDDATA = 0xFF;

}

int write_output_portd(unsigned int which_bit)
{
 unsigned char temp;

 if (which_bit > 7) return -1;

 temp = 0x01 << which_bit;
 PDDATA = PDDATA & ~temp;
 if (temp == 0) return 0;
 else return 1;
}

int main(void)
{
 unsigned int out_led;
 int result;
```

```
  init_portd();


  while(1){
    printf("\nWhich led [range 0 ->7] shall we turn on?\n");
    scanf("%d", &out_led);
    result = write_output_portd(out_led);
    if(result < 0){
      printf("Invalid led number.\n");
      continue;
    }
  }
}
```

5 - 2.2 Example Program #2 - Read Port D

This program asks the user which switch to read and outputs the result to the console.

A listing of read_switch.c follows:

```
/*
  filename: read_switch.c
  programmer: R.A. Sevenich
*/
```

```
#include <asm/MC68EZ328.h>

void init_portd(void)
{
  /* select port D bits for i/o */
  PDSEL = 0xFF ;
  /* set bits 0-7 as input */
  PDDIR = 0x00 ;
}

int read_input_portd(unsigned int which_bit)
{
 unsigned char temp;

 if (which_bit > 7) return -1;

 temp = 0x01 << which_bit;
 temp = PDDATA & temp;
 if (temp == 0) return 0;
 else return 1;
```

```
}

int main(void)
{
  unsigned int in_switch;
  int result;

  init_portd();

  while(1){
    printf("\nWhich switch [range 0 ->7] shall we read?\n");
    scanf("%d", &in_switch);
    result = read_input_portd(in_switch);
    if(result < 0){
      printf("Invalid switch number.\n");
      continue;
    }
    if(result == 0) printf("Switch is ON.\n");
    else printf("Switch is OFF.\n");
  }
}
```

5 - 3 Running the Example Programs

Let's explore how to run a program in a testing environment, using write_led.c as our test candidate. Then, assuming that the test is successful, we'll look at adding the program to the FLASH ROM image and running it from there. This scenario could then be used for any application program destined for the uCsimm/uClinux target.

5 - 3.1 Running from a Testing Environment

The major reason for including the ethernet port on the uCsimm is to provide that capability for the embedded application, but it is certainly convenient for development as well. We saw earlier how we could nfs mount our working directory, existing on our Linux workstation, and thereby incorporate it into the uCsimm/uClinux file hierarchy. This means that we can run a compiled example program from that directory on our uCsimm. Hence, we can test typical programs before moving them to the FLASH ROM.

Let's assume that our working directory is /home/jeff/kit on the development station and that this has been successfully mounted on the target hierarchy at mount point /usr (consistent with Section 5.3 of Chapter 3). Consider testing *write_led.c* (see Section 2.1 of this chapter). Remember that your linux box plays two roles

- with minicom it is the serial terminal for the target
- otherwise it is the cross development platform

Of course, in the X Windows system you can have a window for each - most convenient.

Here, then, are the necessary steps to test *write_led.c*:

- on your development station use an editor to enter *write_led.c* into your working directory (e.g. resulting in */home/jeff/kit/write_led.c*) - or maybe the instructor has a copy on a floppy so you can save some typing

- on your development station, compile *write_led.c* from within its directory via *m68k-pic-coff-gcc write_led.c -o write_led*, resulting in the desired executable i.e. */home/jeff/kit/write_led*
- from your serial terminal, run the program on the target via */usr/write_led*

Of course, your working directory will likely be different than */home/jeff/kit*, but you get the idea.

5 - 3.2 Running in the FLASH ROM Environment

Ultimately we think of running the uCsimm/uClinux in an embedded application, with no connection to a development station. It may still be connected to Linux host via serial connection (a terminal) or via ethernet for some interaction related to the mission of the application (e.g. control room data display, supervisory control, peer to peer communication ...). In any case, we want our application resident on the uCsimm/uClinux target.

It is convenient to pick up where we left off in the prior section and assume that our tested executable program exists as /home/jeff/kit/write_led. We will also assume that /home/jeff/kit is our present working directory when showing command line entries. We perform the following steps to incorporate this executable in the FLASH ROM image:

- copy the executable program *write_led* to location */home/jeff/kit/romdisk/bin* via *cp write_led romdisk/bin/*
- do a make to build the new *image.bin* containing *write_led* via *make*
- load it into the uCsimm's FLASH ROM via *flashloader image.bin*

This last command will also reboot the new uClinux image on the uCsimm. You can then execute your example program on the uClinux command line by entering */bin/write_led*.

Note: When uClinux boots, its last step is to execute */etc/rc*. If your final configuration needs to start your application during the boot process you can invoke the corresponding executable from within */etc/rc*. On your development station this is stored in your working directory at romdisk/etc/rc. You can edit it appropriately there, but do so before the *make* step so it ends up in *image.bin*. An approach of this sort would be needed if the product were an embedded system with no serial terminal connected.

5 - 4 Primitives for an I/O Control Language

5 - 4.1 The State Machine Engine

The preceding example programs show how to manipulate our simple I/O setup. It's nevertheless good to get some hands-on practice and perhaps pick up a few other interesting ideas. What we'll do here is develop an I/O control language based on some simple ideas. The approach is used in practice, but is not widely known as an approach to industrial control - so it may be new to you. However, it is based on the well known state machine. The development will require that we implement a few reusable primitive C functions for these tasks:

- read an I/O point from an image to see if it is ON [bit_on()]
- read an I/O point from an image to see if it is OFF [bit_off()]
- write a bit to 1 in an image [wr_bit_on()]

Traditionally, relay ladder logic has been used to program PLC's (Programmable Logic Controllers); although, the underlying control language may now be hidden under a GUI. A cleaner, more maintainable approach is to base the control language on a set of concurrent, generalized state machines (GSM's). One can also find Petri Net based approaches. For our purposes here, we'll look at an extremely simplified control language using the GSM concept and apply it to our system with its I/O limited to Port D.

Like the classical Finite State Machine (FSM) the GSM has

- a finite number of states
- rules for making a transition to a different state
- a specified start state

Perhaps the major difference is that the GSM has access to memory for storage of information.

The control algorithm in pseudocode form is:

```
New_state = start_state

While (process control is active) {

    Clear write_image

    Read all I/O values and store into read_image

    Current_state = New_state

    Parse Current_state in each GSM basing decisions on read_image and any other
    relevant information. Any decisions to write outputs are stored in write_image
    for later transfer to the actual output. If there is a transition to a different state,
    call that New_state.

    Transfer write_image to outputs

}
```

The algorithmic step which parses the Current_state is actually executing the GSM based control language. This can consist of multiple (concurrent) GSM's, each with many states. For simplicity, we'll show pseudocode for a single, simple two state GSM:

- start_state: Turn on the manual_mode panel light. If the auto_mode switch gets turned on, go to the run_state.
- run_state: Turn on the run_mode panel light. Turn on the conveyor_belt motor. If the auto_mode switch gets turned off, go to the start_state.

Although our system is not appropriately connected to such field wiring, we pretend it is and assume connection to our Port D with the following I/O assignment:

TABLE 5. I/O assignments

Port D bit number	I/O type	I/O name
0	input	auto_mode switch
4	output	manual_mode panel light
5	output	auto_mode panel light
6	output	conveyor_belt motor

Our C language GSM might look like this:

```
switch(current_state) {
  case 0: /* start_state */
    wr_bit_on(4);
    if (bit_on(0)) current_state = 1;
  case 1: /* run_state */
    wr_bit_on(5);
    wr_bit_on(6);
    if (bit_off(0)) current_state = 0;
}
```

Remember, this is inserted within the control algorithm shown earlier.

5 - 4.2 Broadening the State Engine Scope

Our interest here is in this chapter is in examples using the parallel I/O. Further, in looking at today's market for handheld devices that sort of application is what comes first to mind looking at the uCsimm/uClinux system. Nevertheless, it should be apparent that our state engine could easily be broadened in scope.

One can introduce other variables that would be useful, e.g.

- elapsed time
- calendar time
- analog variables such as temperature, pressure
- boolean flags
- general purpose variables

Additional primitive functions dealing with these quantities would also be added to the language such as an elapsed time conditional etc.

Additional hardware support would include circuitry to create a PLC-like capabilities. The uCsimm's general purpose I/O would be divided into some number of address lines, perhaps 8 data lines, and a few lines for control (e.g. read/write capabilities). This 3 bus structure would then be used to access addressable I/O boards, perhaps daisy chained. Such addressable I/O boards exist and not only handle digital I/O, but also analog I/O via A/D and D/A conversion modules. Throw in the uCsimm's existing ethernet capabilities, and you could have a collection of such systems to control a factory and communicate to a centralized control room and with each other via TCP/IP. Classical control algorithms such as PID control are straightforward to incorporate. The LCD panel capabilities would also be of interest for localized display, but then one loses a significant fraction of the parallel I/O for the address, data, and control line capabilities mentioned earlier. One would guess that there is a 144-pin dimm version on the way, where parallel I/O will perhaps exceed the current uCsimm's 21 points.

5 - 5 Activities / Exercises

5 - 5.1 Test the two example programs

Test the two example programs (see Sections 2.1 and 2.2) per the description in Section 3.1.

5 - 5.2 Modify write_led.c

Modify this program to write bits on or off.

5 - 5.3 Incorporate a program into the FLASH ROM

Using the description in Section 3.2, pick one of the example programs to incorporate into the FLASH ROM. Do not set it up to execute during boot.

5 - 5.4 Mini Project per Section 4 - a State Engine

Phase 1

Design, implement, and test the three primitives:

- bit_on()
- bit_off()
- wr_bit_on()

You might begin by defining prototypes for these functions e.g. what are their arguments and return parameters?

The two example programs form Section 2 should be helpful.

Phase 2

Implement and test the control algorithm. A top down design with stubbed functions might be a good starting place. Testing will require a GSM. You might just use the one from Section 4. Keep the GSM as a separate function for modularity.

5 - 5.5 An Example Using your State Engine

The prior mini project has essentially produced a state engine. Choose an example application of your own (or an example from your instructor) and design the appropriate GSM. Insert it in your state engine from the prior subsection and get it working.

CHAPTER 6

Using the uCsimm Ethernet Capability

6 - 1 Introduction

One of the attractive features of the uCsimm/uClinux combination is that it provides the necessary hardware and software for ethernet connectivity. An example application might have a server program running on a central Linux workstation and a number of uCsimm's distributed throughout a factory, all as clients of that single server. The server might display data received from the distributed uCsimm's and provide some sort of overarching supervisory control. The whole system would be closed to the outside world, a dedicated LAN.

It is also easy to think of running the server on the uCsimm. For example, a uCsimm might be connected to data gathering sensors at a remote location. This data might be routinely harvested through an Internet connection. If we connect a uCsimm to the Internet at large, there are some things to keep in mind; because opening any embedded system to the outside world requires care - both for security and performance reasons. By itself, uClinux is soft real-time not hard, which has some performance consequences. If the uCsimm/uClinux system is open to the Internet, contact from the outside could monopolize the CPU. A malicious, but simple Denial of Service attack is quite possible. Such an attack would keep the CPU busy, preventing it from other tasks. Similarly, if Internet traffic to the uCsimm was merely benignly high, performance would still suffer. If the embedded application cannot risk such performance degradation, uClinux by itself may not be

the proper choice. However, it should be noted that there are several versions of real-time Linux (one being RTLinux) and that there are RTLinux extensions available for uClinux. This topic is outside the scope of this course, but would be a good candidate for a following project. Further information can be found in the uClinux email archives.

To explore the uCsimm's ethernet capabilities we will use the socket mechanism, which supports the client/server interaction either locally on a single machine or over an internet e.g. over a local LAN. Although this course assumes that the participants have at least a slight acquaintance with Linux and are familiar with C programming; it is quite possible that sockets are new to some. Consequently, we will spend some time establishing familiarity with that topic. We will not give a thorough overview of socket programming, but will focus more narrowly on the basics we need to get a few useful examples up and running.

Sockets provide a special kind of interprocess communication that emphasizes the distinction between a server process and a client process. The server and client may be on the same machine or on different machines connected via ethernet. The server responds to requests from the client or perhaps from many clients. Sockets underlie such capabilities as printing across a network, ftp, remote login, etc.

The socket API allows one to develop applications based on various protocols such as

- TCP/IP
- UDP/IP
- AppleTalk
- Novell IPX

Internet protocols include both datagram (e.g. UDP/IP) and stream types (e.g. TCP/IP), but we will discuss only the latter.

6 - 2 The Socket Mechanism using TCP/IP

Because the socket mechanism supports various protocols and provides significant flexibility the setup and boilerplate are relatively complex. Nevertheless, the underlying tasks to be performed are straightforward. We'll demonstrate this with

pseudocode for a simple server and then for a simple client. You can inspect these to see how the simple server and client interact.

6 - 2.1 Pseudocode for a Simple Server

Note that the order in the following pseudocode is significant.

- Create an unnamed socket
- Name the socket (define protocol etc.)
- Create a queue for holding client requests
- Accept a connection upon receiving a request from a client
- Communicate (read/write) with the client as needed
- When done, close any open sockets

6 - 2.2 Pseudocode for a Simple Client

Note that the order in the following pseudocode is significant.

- Create an unnamed socket
- Name the socket to agree with the server choices
- Request connection from the server (upon success, the connection is established)
- Communicate (write/read) with the server as needed
- When done, close the socket

6 - 2.3 Supporting details for the pseudocode steps

Our goal in this section is to flesh out the pseudocode and move toward a server program and a client program. The initial client/server interaction will be on the target machine alone, but will use its network capabilities. Before we actually treat each of the steps in the pseudocode, let's write down the include files and declarations that will support both programs:

```
#include <sys/types.h>

#include <sys/socket.h>

#include <stdio.h>
```

```
#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>
```

Next we'll discuss the bulleted items in the server pseudocode and then look at any bulleted items from the client that remain undiscussed.

Create an unnamed socket - server code

This requires that we declare an int and then make a *socket* call with the return value assigned to the declared int:

```
int serv_sock_desc;

serv_sock_desc = socket(AF_INET, SOCK_STREAM, 0);
```

The socket system call is rather like the *open* file system call in that it returns an integer (socket descriptor) to be used to access the socket in the remainder of the program. Note that it takes three arguments:

- a domain
- a socket type
- a protocol

We will not survey the possible choices for these arguments, but merely point out that we have chosen

- the internet address family as the domain, by entering AF_INET
- the continuous byte stream type, by entering SOCK_STREAM
- the default prototype (TCP) by entering 0

Name the socket - server code

This consists of declaring a structure to hold the name information, filling out that structure, and then making the *bind* call to bind the structure to the formerly unnamed socket. In the following code fragment, please notice the use of the function *inet_aton* in place of the older *inet_addr*. The latter function had some problems requiring awkward workarounds in certain cases and use of *inet_aton* is

therefore encouraged. However, note that *inet_aton* returns 0 on error, atypical of most library functions. Here is the corresponding code fragment:

```
int serv_len, result;

struct sockaddr_in serv_addr;

serv_addr.sin_family = AF_INET;

result = inet_aton("127.0.0.1", &serv_addr.sin_addr);

if (result == 0){

  perror("inet_aton error");

  exit(1);

}

serv_addr.sin_port = htons(4242);

serv_len = sizeof(serv_addr);

bind(serv_sock_desc, (struct sockaddr *)&serv_addr, serv_len);
```

The *bind* function takes three arguments:

- the socket descriptor returned by an earlier call to socket
- the address of the sockaddr structure holding the name information
- the length of that name structure

The name information in the sockaddr structure includes:

- the domain (AF_INET)
- the internet address of the socket (127.0.0.1)
- the port number to be bound to the socket

In our case, the communication will be local so the internet address is chosen as 127.0.0.1. We've chosen an arbitrary port number, 4242. Port numbers below 1024 are reserved for system use and other higher numbers above that, which have been

registered for use, should be avoided (see */etc/services*). The function *htons*, used in setting the port address, is one of a small family of functions used to convert architecture dependent byte order to the conventional order used by networks. This deals with the problem faced by two computers of different endianness communicating across the network (e.g. common Intel and Motorola chips).

Create a queue for client requests - server code

The listen call creates a queue for holding pending client requests.

```
listen(serv_sock_desc, 1);
```

The two arguments to listen are

- the socket descriptor for the socket being listened to (serv_sock_desc)
- the queue length desired - arbitrary, we'll use 1 for our example, but something like 5 would be more typical

Accept a connection - server code

The default behavior of accept is to block until a client connects.

```
int cli_sock_desc, cli_len;

struct sockaddr_in cli_addr;

cli_len = sizeof(cli_addr);

cli_sock_desc = accept(serv_sock_addr, (struct sockaddr *)&cli_addr,

                       &cli_len);
```

Note that accept return a new socket descriptor by using the existing socket (identified by *serv_sock_addr*) as a template for the type. This is reasonable since a server can connect to multiple clients. The *accept* function has three arguments:

- the original socket, to use as a template for the type
- a reference to the *sockaddr* struct for the new socket (it receives the address of the calling client)
- the expected address length (gets reset to the actual length)

Communicate with the client - server code

Communication can be carried out with read and write calls. For example, to read a character from the client and write it back, we might have:

```
char ch;

read(cli_sock_desc, &ch, 1);

write(cli_sock_desc, &ch, 1);
```

Note that we use the socket descriptor returned by an earlier accept call.

Close the socket - server code

This is simply:

```
close(cli_sock_desc);

close(serv_sock_desc);
```

If we now look at the client we note that it has very similar pseudocode except that it has no listen or accept calls, but does have a need to make a connection request. Let’s go through the client steps now.

Create an unnamed socket - client code

This is basically the same as for the server, with a trivial name change for the socket descriptor:

```
int client_sock_desc;

client_sock_desc = socket(AF_INET, SOCK_STREAM, 0);
```

Name the socket - client code

This is again much like the server:

```
int client_len;

struct sockaddr_in client_addr;
```

```
client_addr.sin_family = AF_INET;
result = inet_aton("127.0.0.1", &client_addr.sin_addr);
if (result == 0){
  perror("inet_aton error");
  exit(1);
}
client_addr.sin_port = htons(4242);
client_len = sizeof(client_addr);
bind(client_sock_desc, (struct sockaddr *)&client_addr, client_len);
```

Request a Connection - client code

If successful, the connection is established.

```
int result;
result = connect(client_sock_desc, (struct sockaddr *)&client_addr, client_len);
```

Here, we would check for failure (result == -1) and deal with it.

Communicate with the server - client code

This would mesh with the server, e.g.

```
char ch;
write(client_sock_desc, &ch, 1);
read(client_sock_desc, &ch, 1);
```

Close the socket - client code

This is, as for the server:

```
close(client_sock_desc);
```

In the next section, we'll assemble these fragments into two programs, one for the server and the other for the client.

6 - 3 A First Example - just the uCsimm

Our first example will have both server and client on a single uCsimm. Further, their communication is unusually short. In typical cases, the server would continue indefinitely, awaiting more client requests. Ultimately, we'll proceed to more typical examples that will have the server and clients on separate machines.

We can build a server and a client based on the discussion in the prior section. In the two following subsections we list such programs. Subsequently we'll reiterate the steps needed top get them up and running on the uCsimm/uClinux target.

6 - 3.1 The server program

```
/*
  filename: serv0.c
  programmer/transcriber: R.A. Sevenich

  * use with cli0.c
  * compile both on same machine
  * execute serv0 first, putting it in the background
    e.g. ./serv0 &
  * then execute cli0 in the foreground
    e.g. ./cli0
```

```
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main()
{
   int serv_sock_desc, serv_len;
   struct sockaddr_in serv_addr;
   int cli_sock_desc, cli_len;
   struct sockaddr_in cli_addr;
   char ch;
   int i, result;

   serv_sock_desc = socket(AF_INET, SOCK_STREAM, 0);
   i = 1;
   setsockopt(serv_sock_desc, SOL_SOCKET,
```

```
        SO_REUSEADDR, &i, sizeof(i));

serv_addr.sin_family = AF_INET;

result = inet_aton("127.0.0.1", &serv_addr.sin_addr);
if (result == 0){
  perror("inet_aton error");
  exit(0);
}

serv_addr.sin_port = htons(4242);
serv_len = sizeof(serv_addr);
bind(serv_sock_desc, (struct sockaddr *)&serv_addr, serv_len);

listen(serv_sock_desc, 1);
printf("Server awaiting client request\n");
cli_len = sizeof(cli_addr);
cli_sock_desc = accept(serv_sock_desc,
            (struct sockaddr *)&cli_addr, &cli_len);
read(cli_sock_desc, &ch, 1);
write(cli_sock_desc, &ch, 1);
```

```
    close(cli_sock_desc);
    close(serv_sock_desc);
}
```

6 - 3.2 The client program

```
/*
 filename: cli0.c
 programmer/transcriber: R.A. Sevenich

 * use with serv0.c
 * compile both on same machine
 * execute serv0 first, putting it in the background
   e.g. ./serv0 &
 * then execute cli0 in the foreground
   e.g. ./cli0
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
```

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main()
{
   int client_sock_desc, client_len;
   struct sockaddr_in client_addr;
   int result;
   char ch = '3';

   client_sock_desc = socket(AF_INET, SOCK_STREAM, 0);
   client_addr.sin_family = AF_INET;
   result = inet_aton("127.0.0.1", &client_addr.sin_addr);
   if (result == 0){
     perror("inet_aton error");
     exit(1);
   }
   client_addr.sin_port = htons(4242);
   client_len = sizeof(client_addr);
   bind(client_sock_desc, (struct sockaddr *)&client_addr, client_len);
```

```
    result = connect(client_sock_desc, (struct sockaddr *)&client_addr,
                client_len);

    if(result == -1) {
       perror("Client cannot connect");
       exit(1);
    }

    write(client_sock_desc, &ch, 1);
    printf("Client sends <%c> to server, ", ch);
    ch = '5';
    read(client_sock_desc, &ch, 1);
    printf("and gets <%c> back.\n", ch);
    close(client_sock_desc);
    exit(0);
}
```

6 - 3.3 Running the programs on the uCsimm

We assume here that your Linux workstation is up and running with the uCsimm connected serially and by ethernet. Further, we assume that uClinux has booted on the uCsimm and has successfully done an nfs mount of the Linux workstation's working directory (e.g. /home/jeff/kit).

To get this example working, you would enter the previous server and client programs into your working directory on your Linux workstation. Let's call them serv0.c and cli0.c. Then you would compile them via

```
m68k-pic-coff-gcc -O2 serv0.c -o serv0.x

m68k-pic-coff-gcc -O2 cli0.c -o cli0.x
```

Note the use of the '-O2' (second level optimization) in the compilation. Without some level of optimization the compiler encounters some in line functions which it fails to expand and sees the problem as unresolved references. In this particular case, the problem arises in parsing <asm/byteorder.h>. Using level 2 (-O2) is sometimes recommended and easy to remember - the compiler needs extra oxygen to expand the in line functions.

These programs are to be run on the target, so run them on the serial terminal window established with minicom. Assuming you have moved to the proper directory (available on the target via the nfs mounted directory), run your server and client by entering:

```
./serv0.x &

./cli0.x
```

The first line starts the server and puts it in the background, awaiting a client request. The server's *accept* function blocks, so the server waits patiently. The second line starts the client, so the two programs can

- establish their connection
- carry out their communication and exit

6 - 4 A Second Example - uCsimm and Workstation as client/server

We next move the server so it executes from the Linux workstation, while the client will execute on the uCsimm. We'll keep the client/server interaction unrealistically simple so our focus will be only on the salient changes.

6 - 4.1 Changes to the server program

The only change in the server is the line

```
result = inet_aton("127.0.0.1", &serv_addr.sin_addr);
```

which becomes:

```
result = inet_aton("192.168.1.4", &serv_addr.sin_addr);
```

where 192.168.1.4 is the IP address of the server and will likely need to be changed for your machine.

6 - 4.2 Changes to the client program

Similarly, the only change in the client is the line:

```
result = inet_aton("127.0.0.1", &client_addr.sin_addr);
```

which becomes:

```
result = inet_aton("192.168.1.4", &client_addr.sin_addr);
```

where 192.168.1.4 is again the IP address of the server (*not* of the client) and will likely need to be changed for your setup.

6 - 4.3 Running the server and client programs

Enter the newly modified server and client programs into your working directory on your Linux workstation. Let's call them serv1.c and cli1.c. Compile them via

```
gcc -O2 serv1.c -o serv1.x (remember the server now runs on the workstation)

m68k-pic-coff-gcc -O2 cli1.c -o cli1.x
```

Once these have successfully compiled, do the following

- from your workstation terminal window (not the minicom window), run the server on the workstation by entering *./serv1.x*
- then from your minicom window, run the client on the target by entering *./cli1.x*

You will carry this out in the Activities/Exercises.

6 - 5 Adding some Flexibility and Complexity to the Examples

6 - 5.1 Setting socket options

There is the setsockopt function for setting socket options. It's prototype is:

```
#include <sys/socket.h>

int setsockopt(int socket, int level, int option_name,

        const void *option_value, size_t option_len);
```

We'll not explore the various options, but will consider one useful example.

Linux limits how soon a socket can be reused, with a two minute limit being typical for TCP. The following removes that restriction:

```
int i;

i = 1;

setsocketopt(serv_sock_desc, SOL_SOCKET, SO_REUSEADDR,

        &i, sizeof(i));
```

where

- the first argument specifies the socket whose options are to be set
- the second specifies that a generic socket is being set
- the third argument specifies the option to be set
- the fourth argument is a pointer to a non-zero integer which, in this case, turns the SO_REUSEADDR option on
- the fifth argument is the size of the value referenced by the fourth argument

6 - 5.2 Alternative to hard coded IP addresses

The IP addresses in our code examples were hard coded and in certain embedded environments this may be what you want. However, there is an alternative.

The server may wish to accept incoming connections on any address it has for the local connection - either for code portability or because it just doesn't care. If the server's sin_addr field of its sockaddr_in struct has each byte filled with zero, this is a don't care indication. For example, replace:

```
result = inet_aton("192.168.1.4", &serv_addr.sin_addr);
```

by

```
memset(&serv_addr.sin_addr, 0, sizeof(serv_addr.sin_addr))
```

This sort of don't care idea can also be used for the port number, but is less likely to be useful.

6 - 5.3 Serving Multiple Clients using Select

In our examples so far, the server blocks during the function *accept*; i.e., it waits for a client connection request. If the server's only job is to be a server, that is fine. However, if the server has other things to do this behavior is inefficient. In this section, we'll explore implementing non blocking behavior. To do so we'll introduce the *select* system call. In the 2.2 kernel version, *select* became deprecated in favor of *poll*. Currently uClinux is derived from the earlier kernel version so we'll stick with *select* to allow our material to apply to both target and development workstation.

The *select* system call provides this behavior:

- *select* can watch sets of device/file descriptors waiting for activity
- *select* returns immediately if any of those devices/files need attention and indicates which ones those are
- *select* blocks if none of the devices/files need attention, but ceases blocking when any of them become active
- however, a timeout can be assigned to the blocking to achieve a sort of non blocking

The select prototype is:

```
#include <sys/time.h>

#include <sys/types.h>
```

```
#include <unistd.h>

int select( int nfds, fd_set *rdfs, fd_set *wrfds, fd_set *exfds,

        struct timeval *timeout);
```

where

- nfds specifies that the descriptors to be watched are in the range 0 to nfds - 1
- rfds specifies a set of descriptors to be watched for input activity
- wrfds specifies a set of descriptors to be watched for output activity
- exfds specifies a set of descriptors to be watched for exception/error conditions
- *timeout* specifies how long *select* will await activity; a zero value specifies that it will wait forever

If any of the above descriptor sets reference the NULL pointer, that set is not watched.

In addition, various useful macros are provided, as follows:

- FD_ZERO(fd_set *fdset); - initializes an fd_set to empty
- FD_CLR(int fd, fd_set *fdset); - clears element fd from an fd_set
- FD_SET(int fd, fd_set *fdset); - sets element fd into an fd_set
- FD_ISSET(int fd, fd_set *fdset); - returns non zero if fd is a member of fd_set
- FD_SETSIZE - fills in the appropriate nfds value

6 - 5.4 An example using select

As an example let’s set up a server on the Linux workstation to use select to watch for multiple clients. Here is the code:

```
/*

  filename: serv3.c

  programmer/transcriber: R.A. Sevenich
```

```
 * use with cli1.c
 * compile both on same machine or different machines
 * execute serv3 first, putting it in the background
   if any clients are on the same machine
   e.g. ./serv3 &
 * then execute multiple cli1's on its machine
   e.g. ./cli1 & ./cli1 & ./cli1
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

#define true 1
#define false 0

int main()
```

```
{
  int serv_sock_desc, serv_len;
  struct sockaddr_in serv_addr;
  int cli_sock_desc, cli_len;
  struct sockaddr_in cli_addr;
  char ch;
  int i, res;

  fd_set input_set, watch_input_set;
  int desc, nbr;
  int watching;

  serv_sock_desc = socket(AF_INET, SOCK_STREAM, 0);


  i = 1;
  setsockopt(serv_sock_desc, SOL_SOCKET,
        SO_REUSEADDR, &i, sizeof(i));

  serv_addr.sin_family = AF_INET;
```

```
memset(&serv_addr.sin_addr, 0, sizeof(serv_addr.sin_addr));


serv_addr.sin_port = htons(4242);
serv_len = sizeof(serv_addr);
bind(serv_sock_desc, (struct sockaddr *)&serv_addr, serv_len);
listen(serv_sock_desc, 5);
watching = true;
FD_ZERO(&input_set);
FD_SET(serv_sock_desc, &input_set);


while(watching){
 printf("Server awaiting multiple client requests.\n");
 watch_input_set = input_set;
 res = select(FD_SETSIZE, &watch_input_set, (fd_set *)0,
            (fd_set *)0, (struct timeval *)0);
 if (res < 0){
  perror("select failed");
  exit(1);
 }
 for(desc = 0; desc < FD_SETSIZE; desc++){
  if(FD_ISSET(desc, &watch_input_set)){
```

```
/* this clause is for a new client */
if(desc == serv_sock_desc){
        cli_len = sizeof(cli_addr);
        cli_sock_desc = accept(serv_sock_desc,
                (struct sockaddr *)&cli_addr, &cli_len);
        FD_SET(cli_sock_desc, &input_set);
  printf("New client with descriptor = %d\n", cli_sock_desc);
      }

/* this clause is for an existing client */
else{
  ioctl (desc, FIONREAD, &nbr);

  /* this clause finds a client who did a close operation */
  if (nbr == 0) {
    close(desc);
    FD_CLR(desc, &input_set);
    printf("Closing out client with descriptor = %d\n", desc);
  }
```

```
            /* this clause finds a client doing a write operation */
            sleep(3);
        read(desc, &ch, 1);
        printf("Server dealing with client (descriptor = %d).\n", desc);
            write(desc, &ch, 1);
          }
        }
      }/* end of the for loop */
    }/* end of the while loop */

    /* program, as is, won't get here */
    close(serv_sock_desc);
    exit(0);
  }
```

6 - 6 Activities / Exercises

6 - 6.1 Server and client on a single uCsimm

Modify the server program from section 3.1 so that the server

- remains in an eternal loop
- changes the character sent by the client before sending it back

Compile and run the client and server programs on your uCsimm.

6 - 6.2 Server on Linux Workstation, client on uCsimm

Implement the modifications to server and client as described in Section 4.

6 - 6.3 Server on uCsimm, client on Linux Workstation

Reverse the roles from Section 6.2

6 - 6.4 Change server to not care which of its IP addresses is used for the incoming connection

See Section 5.2.

6 - 6.5 uCsimm to uCsimm

If you are taking this course with other participants so another uCsimm is available, partner up with someone and attempt to design, implement, and test a client/server interaction between the two uCsimm's.

6 - 6.6 Using select - blocking

Give a pseudocode design for the while loop of the code of section 5.4. It contains a for loop which investigates several different cases - pay particular attention to these. Finally, implement the program and try it out. The code from Section 3.2 should work fine as the client for this server. You might start several such servers e.g. at the target's command line with something like:

```
./cli0.x & ./cli0.x & ./cli0.x
```

Explain the output in detail.

6 - 6.7 Using select - non blocking

Give the select a timeout value and investigate the nonblocking behavior by giving the program something else to do e.g. allow the user to decide to terminate the program.

6 - 7 References

Among the classic references by Stevens, we suggest:

W. Richard Stevens, *UNIX Network Programming*, Prentice Hall (1990).

W. Richard Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, Addison Wesley Longman (1994).

The following two books are somewhat general purpose, but have good chapters on socket programming with the emphasis on TCP/IP:

Michael K. Johnson and Erik W. Troan, *Linux Application Development*, Addison Wesley Longman (1998).

Neil Matthew and Richard Stones, *Beginning Linux Programming*, 2nd Edition, WROX (1999).

CHAPTER 7

Making Your Own Embeddable Linux

7 - 1 Introduction

After using a Linux distribution, some of us may wonder how difficult it would be to build our own. In this chapter we will do just that. Our goal will be to build a small version e.g. no X Windows System. There are a number of useful references and a few software packages intended to help us reach our goal of a small system. Perhaps the most commonly used software packages are *Busybox* and *Tinylogin*. Each is intended to be a small replacement for a selection of larger packages, saving considerable space while providing adequate functionality.

While all the sources cited in the reference section are helpful, the one of most direct benefit was *Building Tiny Linux Systems with Busybox: Part 2* by Bruce Perens. At the time of this writing, *Part 2* was not yet published, but Bruce was gracious enough to make a draft available, thereby exemplifying the open attitude that often accompanies open source. Installation documentation for *Tinylogin* is currently not as useful, but we were able to get it up and running.

Neither *Busybox* nor *Tinylogin* has reached version 1.0. They are functional and useful, but we should expect changes. For example, *Busybox* does not depend on the Name Service Switch scheme, while *Tinylogin* does. Reportedly, *Tinylogin* will also move away from this scheme. The bottom line is that this document is doomed to obsolescence and that you should use it with care.

To have a functional Linux system we need two pieces of software:

- a Linux kernel
- a root file system

In building a small Linux, we could start from an existing Linux version and cut out parts we don't need and, where possible, replace needed parts by smaller versions which may sacrifice some functionality, but will otherwise suffice. Perhaps the most difficult obstacle is interdependencies that may be hidden or unexpected i.e. we may remove some functionality that we think we don't need only to find that something else we require is now broken because of a subtle dependency on the functionality we just threw away. Here we are especially fortunate that others have blazed a trail. In particular, we find the *Busybox* package which is a relatively complete and miniaturized tool set for populating the needed root file system. This allows us a different tack; i.e., we start with a very small, but functional system with a kernel we compiled and a root file system based on *Busybox*. Then we can add functionality, item by item.

In this chapter we will explore two examples, the first results in a small Linux, resident on a floppy drive. This example uses both the *Busybox* tool set (static library inclusion) installed within a root file system and a kernel image. Upon boot it installs itself to RAM. The second example results in a small Linux resident on either an extra partition or, equivalently, a zip disk. The second example includes both *Busybox* and *Tinylogin*. Dynamic library linkage is used here.

At first glance, the examples don't seem 'embedded'. However, either can convert a machine to a purpose normally done by an embedded machine. For example, the Linux on a floppy can provide not only a rescue disk, but alternatively convert an old 486 to

- a router - see for example, http://www.linuxrouter.org and http://www.fresco.org
- a dedicated firewall - see for example, http://www.zelow.no/floppyfw

and the like. Further, these examples can provide a staging area for an embedded system, as is done with uClinux.

In Chapters 8 and 9 the *Embedix/SDK* is explored. This package also uses *Busybox* and *Tinylogin*, but has made the process of tailoring an embedded system more automated and much easier. The current chapter will provide some insight into the innards of the *Embedix/SDK*, but doesn't stray far from the minimal root file system in which *Busybox* and *Tinylogin* reside. Adding functionality to that minimal core is

a bit tricky and the ease of use provided by the *Embedix/SDK* takes the pain out of the process.

7 - 2 Busybox

7 - 2.1 What is Busybox?

Busybox was originally implemented by Bruce Perens for the Debian GNU/Linux distribution. It helped provide a complete, bootable system on a single floppy to serve as a rescue disk and an installer for the full blown Debian system. Many others have subsequently contributed to Busybox, which is currently maintained by Eric Andersen. The rise of embedded Linux has given it an exciting new application area where it is deployed in many such systems. It provides over 100 command line tools including ls, cat, chmod, dd, gzip, tar, and so on. The current version as of this writing is 0.47. To see the complete list of commands provided, we can look at this excerpt from the Busybox Config.h:

```
//

// BusyBox Applications

#define BB_AR

#define BB_BASENAME

#define BB_CAT

#define BB_CHMOD_CHOWN_CHGRP

#define BB_CHROOT

#define BB_CHVT

#define BB_CLEAR

#define BB_CP_MV

#define BB_CUT

#define BB_DATE
```

```
#define BB_DC

#define BB_DD

#define BB_DEALLOCVT

#define BB_DF

#define BB_DIRNAME

#define BB_DMESG

#define BB_DOS2UNIX

#define BB_DUTMP

#define BB_DU

#define BB_DUMPKMAP

#define BB_ECHO

#define BB_EXPR

#define BB_FBSET

#define BB_FDFLUSH

#define BB_FIND

#define BB_FREE

#define BB_FREERAMDISK

#define BB_FSCK_MINIX

#define BB_GETOPT

#define BB_GREP

#define BB_GUNZIP

#define BB_GZIP

#define BB_HALT
```

```
#define BB_HEAD
#define BB_HOSTID
#define BB_HOSTNAME
#define BB_ID
#define BB_INIT
#define BB_INSMOD
#define BB_KILL
#define BB_KILLALL
#define BB_LENGTH
#define BB_LN
#define BB_LOADACM
#define BB_LOADFONT
#define BB_LOADKMAP
#define BB_LOGGER
#define BB_LOGNAME
#define BB_LS
#define BB_LSMOD
#define BB_MAKEDEVS
#define BB_MD5SUM
#define BB_MKDIR
#define BB_MKFIFO
#define BB_MKFS_MINIX
#define BB_MKNOD
```

```
#define BB_MKSWAP

#define BB_MKTEMP

#define BB_NC

#define BB_MORE

#define BB_MOUNT

#define BB_MT

#define BB_NSLOOKUP

#define BB_PING

#define BB_POWEROFF

#define BB_PRINTF

#define BB_PS

#define BB_PWD

#define BB_RDATE

#define BB_REBOOT

#define BB_RENICE

#define BB_RESET

#define BB_RM

#define BB_RMDIR

#define BB_RMMOD

#define BB_SED

#define BB_SETKEYCODES

#define BB_SH

#define BB_SLEEP
```

```
#define BB_SORT
#define BB_SWAPONOFF
#define BB_SYNC
#define BB_SYSLOGD
#define BB_TAIL
#define BB_TAR
#define BB_TEE
#define BB_TEST
#define BB_TELNET
#define BB_TOUCH
#define BB_TR
#define BB_TRUE_FALSE
#define BB_TTY
#define BB_UNRPM
#define BB_UPTIME
#define BB_USLEEP
#define BB_WC
#define BB_WGET
#define BB_WHICH
#define BB_WHOAMI
#define BB_UUENCODE
#define BB_UUDECODE
#define BB_UMOUNT
```

```
#define BB_UNIQ

#define BB_UNAME

#define BB_UNIX2DOS

#define BB_UPDATE

#define BB_XARGS

#define BB_YES

// End of Applications List

//
```

Notable in the list, because of their special roles are:

- BB_SH, a tiny shell which will be installed at /bin/sh
- BB_INIT, an init which will be installed at /sbin/init

Busybox uses a clever trick so that it is much smaller than the aggregate size of all the commands it replaces. Every executable command includes a fixed number of bytes constituting a more or less common overhead, perhaps several kilobytes in each instance. However, *Busybox* is a single executable which can be linked to the 100 plus command names. *Busybox* requires from 256 to 500 kilobytes on the IA-32 to support all these commands. The range in size is due to how linkage to certain libraries is established, i.e. dynamic or static. Further, it can be tailored to omit less crucial commands to decrease the binary size, but only slightly.

7 - 2.2 Getting Busybox

Busybox is available from http://busybox.lineo.com as a compressed tarball, slightly smaller than 500 kb for version 0.47. Check with your instructor to see if a local copy is available; otherwise download one for the subsequent section with Activities/Exercises.

7 - 3 Tinylogin

7 - 3.1 What is Tinylogin?

Tinylogin was originally assembled from various other contributors by Sean Bastille. Like *Busybox*, it is now maintained by Eric Andersen. The package provides utilities related to login. As used later in this chapter it supports shadow passwords, the addition and deletion of users etc. To be specific it offers these commands:

- adduser, deluser
- addgroup, delgroup
- login
- su
- sulogin
- passwd
- getty

Like *Busybox*, there is a single executable (*/bin/tinylogin*) and the commands listed above are linked to that, thereby avoiding the overhead required for a collection of individual executables.

7 - 3.2 Getting Tinylogin

Tinylogin is available from http://tinylogin.lineo.com as a compressed tarball, slightly smaller than 90 kb for version 0.78. Check with your instructor to see if a local copy is available; otherwise download one for the subsequent section with Activities/Exercises.

7 - 4 Compiling your new Kernel

For both of our examples, we'll need a kernel image, so we'll cover compiling such in this single section. If you haven't compiled a new kernel before, you will likely find it a daunting task. In fact, although it is straightforward, many things can possibly go awry. Further, there are many different possible initial configurations and it's impractical to cover them all here. Rely on your instructor for help if you aren't experienced with this.

With that said, we are creating an image for either an extra partition or for a floppy, not for your existing work station. Hence, if we're careful and conservative we can make a few mistakes and still not harm your existing work station's boot image or boot process.

Creating the option for a graceful recovery

As root change to the source hierarchy via

```
cd /usr/src
```

and inspect the premises via

```
ls -l
```

You should see the name linux and perhaps other stuff. The name linux is either a link to a directory or a link to a directory. Let's look at each case:

Linux is not a link, it is actually a directory

Take the following steps:

```
mv linux/ orig_linux

tar cvfz linux_orig.tgz orig_linux/

mv orig_linux/ embed_linux

ln -s embed_linux/ linux

cd linux
```

Linux is a link, say to linux-2.2.9, for a specific example

Take the following steps

```
tar cvfz linux-2.2.9.tgz linux-2.2.9/

rm linux

mv linux-2.2.9/ embed_linux
```

```
ln -s embed_linux/ linux

cd linux
```

In either of the above two cases, you are now in the linux directory, which is a link to the original source tree. Further, we have a tarball of the original source tree for later restoration - after we've made our new small kernel image.

Make the new Image

There is a README in /usr/src/linux which is instructive. Essentially, there are four steps to follow:

- make mrproper
- make menuconfig (or variants)
- make dep
- make bzImage

In the make menuconfig step, you'll make the many choices which lead to what will be in your new image. Make choices that will make the kernel as small as possible. You can always go back and add functionality. In either of our examples (extra partition or floppy), do not choose loadable modules. In the floppy example, choose

- RAM disk support (in the Block Devices menu)
- Initial RAM disk (initrd) support (in the Block Devices menu)
- ROM file system support (in the File systems menu)]

The resulting compressed image will be located at

```
/usr/src/embed_linux/arch/i386/boot/bzImage
```

from whence we'll retrieve it when needed - much later.

Restore the original source tree before rebooting!

Here there are two cases, depending on the name of the restoration tarball.

restoration tarball name is linux_orig.tgz

Take these steps

rm linux

tar xvfz linux_orig.tgz

ln -s orig_linux/ linux

restoration tarball name is linux-2.2.9.tgz

rm linux

tar xvfz linux-2.2.9.tgz

ln -s linux-2.2.9/ linux

Need to go around for a second try?

If you build and boot your small Linux and need or want to try to recompile again, the prior evolution has left you with your small linux source tree, *embed_linux*, in /usr/src.To try again, you need to do these steps

note what /usr/src/linux is linked to (let's call it linux_whatever/)

rm linux

ln -s embed_linux/ linux

cd linux

Then make another new image via

- DO NOT make mrproper after the first time, but do
- make menuconfig
- make dep
- make bzImage

Once you have the new image, put things back the way they were i.e.

cd /usr/src

rm linux

```
ln -s linux_whatever/ linux
```

7 - 5 Small Linux on a Floppy

7 - 5.1 Strategy - Floppy

Here we'll establish a working directory and, to have a concrete example, we'll call it 'embed' and assume it's in your home directory hierarchy, somewhere like /home/rsmith/embed. Note that the system, as in Section 7 - 3, will consist of a kernel and a root file system. However, this time our target is a floppy and we'll not include login capabilities; hence, no Tinylogin - just Busybox. Our strategy is as follows:

- create a root file system in the working directory, partly relying on Busybox
- compile an appropriate new kernel image
- convert the root file system to a ROM image
- build a bootstrap image onto the floppy
- copy kernel and ROM file system images onto the floppy
- prepare the configuration file which guides the boot process
- try it out

7 - 5.2 Applying the Strategy - Floppy

We'll organize the strategy in steps to provide a recipe for carrying it out. The order of the steps matters in some cases, but not in others - so following the sequence is safest. For many of the steps you need to work as root; so we'll just work as root throughout and be careful.

Step 1. Become root and create a working directory

For example, after becoming root, enter the command:

```
mkdir /home/rsmith/embed
```

Henceforth, we'll refer to this working directory as *embed/*, i.e. without the full pathname.

Step 2. Copy the Busybox tarball to *embed/* and explode the tarball

With busybox-0.47.tar.gz now in *embed/* enter, for example,

```
cd embed
tar xvfz busybox-0.47.tar.gz
```

thereby creating a new directory within *embed/* i.e.

```
embed/busybox-0.47/
```

Step 3. Build Busybox

This requires these smaller steps:

change to the Busybox directory e.g.

```
cd busybox-0.47/
```

with an editor modify the Makefile so the line

```
DOSTATIC=false
```

is replaced by

```
DOSTATIC=true
```

Now enter

```
make
```

Note that you have created a static-linked version Busybox, appropriate for our floppy target.

Step 4. Install Busybox

Now enter

```
make PREFIX="/embed" install
```

Return to the embed/ directory and do a

```
ls -l
```

and you should see, at this hierarchy level, three directories

```
bin/   sbin/   usr/
```

and a link to /bin/busybox named

```
linuxrc
```

You'll also still see the Busybox tarball and the Busybox directory. These are no longer needed and can be removed. Before doing so check that you still have the Busybox tarball elsewhere in case a second try is needed, then

```
rm -rf busybox*
```

It is worth your time to poke around in this new hierarchy to find where various commands are e.g. find *init*, *tar*, *chmod*, *which*, *chroot*, and so on.

Step 5. Add remaining directories to the root file system

Although embed/ now has three top level directories we need the others. To provide them, enter:

```
mkdir dev etc etc/init.d mnt proc tmp var
```

Next set the permissions appropriately (agreeing with the bin/ sbin /usr)

```
chmod 755 dev etc etc/init.d mnt proc tmp var
```

Step 6. Populate the new dev directory with needed devices

Do this by copying from the existing system's /dev directory:

```
cp -av /dev/tty dev
```

```
cp -av /dev/tty0 dev
```

```
cp -av /dev/console dev
```

```
cp -av /dev/ram0 dev
cp -av /dev/null dev
```

Then set permissions appropriately:

```
chmod 666 dev/*
chmod 600 dev/ram0
```

Step 7. Create some needed files

With an editor, create embed/etc/fstab as follows:

```
proc    /proc        proc    defaults    0    0
none    /var/shm     shm     defaults    0    0
```

Next with your editor, create embed/etc/inittab as follows:

```
::sysinit:/etc/init.d/rcS
::askfirst:/bin/sh
```

Finally, with your editor create embed/etc/init.d/rcS as follows:

```
#! /bin/sh
mount -a                # mount file systems from /etc/fstab
```

Adjust the permissions of each by:

```
chmod 644 embed/etc/fstab
chmod 644 embed/etc/inittab
chmod 744 embed/etc/init.d/rcS
```

Step 8. Generate the ROM file system

If genromfs is not available in your existing system, get the tarball from:

http://ibiblio.org/pub/Linux/system/recovery/

where, at this writing we find:

genromfs-0.3.tar.gz

Put the tarball in the directory containing *embed/* (e.g. /home/rsmith).

Change to the directory containing *embed/* (e.g. /home/rsmith) and explode the compressed tarball e.g.

tar xvfz genromfs-0.3.tar.gz

resulting in the directory *genromfs-0.3*, containing among other entries, the executable *genromfs*.

Still from the directory containing *embed/*, generate the ROM file system image for your root directory via

./genromfs-0.3/genromfs -d embed -f fs

and compress the result via

gzip -9 fs

producing the desired image, *fs.gz*.

Step 9. Build the floppy

If syslinux is not available in your existing system, get the tarball from:

http://ibiblio.org/pub/Linux/system/boot/loaders/

where, at this writing we find:

syslinux-1.48.tar.gz

Put the tarball in the directory containing *embed/* (e.g. /home/rsmith).

Change to the directory containing *embed/* (e.g. /home/rsmith) and explode the compressed tarball e.g.

```
tar xvfz syslinux-1.48.tar.gz
```

resulting in the directory *syslinux-1.48*, containing among other entries, the executable *syslinux*.

Now put a clean MS DOS diskette in your floppy - don't mount it. Install the syslinux bootstrap program, working from *embed/* e.g.

```
./syslinux-1.48/syslinux /dev/fd0
```

Step 10. Configure the floppy

Now mount the floppy e.g.

```
mount -t msdos /dev/fd0 /mnt
```

Copy the compressed kernel image and root file system to the floppy:

```
cp /usr/src/embed_linux/arch/i386/boot/bzImage /mnt/linux
```

```
cp fs.gz /mnt
```

With an editor, create the following configuration file, /mnt/syslinux.cfg:

```
TIMEOUT 50
```

```
DEFAULT linux
```

```
LABEL linux
```

```
KERNEL linux
```

```
APPEND root=/dev/ram0 initrd=fs.gz
```

This file is read and followed when the floppy is used subsequently to boot our new system. It tells syslinux to

- wait 5 seconds (so you can interrupt the default boot by pressing the shift key)
- if not interrupted, boot the default
- let the kernel know that the root is a RAM disk (/dev/ram0)

- load the RAM disk from the compressed ROM file system, fs.gz

7 - 5.3 Trying it out - Floppy

We're now ready to boot the system with the floppy. If your machine's BIOS is not configured to boot from floppy, reconfigure so it is. Now reboot, crossing your fingers if possible. The boot process should first show on the screen a line starting with SYSLINUX giving credit to its author Peter Anvin. It then lists things discovered and accomplished in the boot process. which should then end with the line:

> Please press Enter to activate this console.

Doing so should lead to a boot prompt. You can now explore your minimal system.

On the other hand, if the system does not boot, go through all the steps again for another try.

7 - 6 Small Linux on a Zip Disk or Extra Partition

7 - 6.1 Strategy - Zip Disk or Extra Partition

As mentioned earlier, a Linux system can be considered as consisting of a kernel and a root file system, populated appropriately. We'll build our second small example Linux system on a zip disk for our Linux workstation. The difference between using an extra partition on the hard drive and a zip disk are trivial. The machine at hand has a zip drive, saving the re partitioning of the hard drive.

Our strategy will consist of carrying out the following sequential steps:

- mounting the zip drive so we can access it
- creating a root file system on the zip disk
- appropriately populating that file system using *Busybox* and *Tinylogin*
- ensuring that, when deployed, the system will provide a proper boot, a login, and a usable shell
- compile an appropriate new kernel image
- modifying lilo.conf to support the option of booting our new system
- trying it out

7 - 6.2 Applying the Strategy - Zip Disk

We'll organize the strategy in steps to provide a recipe for carrying it out, just as we did for the floppy. Although some of the steps are identical as those for the floppy build, others are slightly or totally different. We'll list all the steps rather than jumping back and forth between sections. The order of the steps matters in some cases, but not in others - so following the sequence is safest. For many of the steps you need to work as root; so we'll just work as root throughout and be careful.

We assume that your zip disk has an ext2 file system. Let's create a directory as a mount point and do the mount e.g.

```
mkdir /zipdisk

mount -t ext2 /dev/hdc /zipdisk
```

where /dev/hdc is this author's device name for the zip drive (yours will likely be different).

Step 1. Become root

Step 2. Copy the Busybox and Tinylogin tarballs to /zipdisk and explode the tarballs. For example, with busybox-0.47.tar.gz and tinylogin-0.78.tar.gz now in */zipdisk* enter:

```
cd /zipdisk

tar xvfz busybox-0.47.tar.gz

tar xvfz tinylogin-0.78.tar.gz
```

thereby creating new directories

```
/zipdisk/busybox-0.47/

/zipdisk/tinylogin-0.78/
```

Step 3. Make and install Busybox

Change to the Busybox directory and do the make e.g.

```
cd /zipdisk/busybox-0.47/

make
```

Now do the install

```
make PREFIX=”/zipdisk” install
```

Step 4. Make and install Tinylogin

Change to the Tinylogin directory and do the make e.g.

```
cd /zipdisk/tinylogin=0.78/

make
```

Now do the install

```
make PREFIX=”/zipdisk” install
```

Step 5. Remove what’s no longer unneeded

The Busybox tarball, the busybox-0.47/ directory, the Tinylogin tarball, and the tinylogin-0.78/ directory are no longer needed. However, make sure you have saved copies of both tarballs elsewhere. Now remove what’s no longer needed:

```
cd /zipdisk

rm -rf busy*

rm -rf tiny*
```

Check what remains with ls -l. You should see, at this hierarchy level, three directories

```
bin/   sbin/   usr/
```

and a link to /bin/busybox named

```
linuxrc
```

Step 6. Add remaining directories to the root file system

Although /extra now has three top level directories we need the others. To provide them, enter:

```
mkdir dev etc etc/init.d home lib mnt proc root tmp usr/lib var
```

Next set the permissions appropriately (agreeing with the bin/ sbin /usr)

```
chmod 755 dev etc etc/init.d home lib mnt proc root tmp usr/lib var
```

Step 7. Populate the new dev directory with needed devices

Do this by copying from the existing system's /dev directory:

```
cp -av /dev/tty dev

cp -av /dev/tty0 dev

cp -av /dev/tty1 dev

cp -av /dev/console dev

cp -av /dev/hdc dev ... or whatever is your zip drive

cp -av /dev/hda3 dev ... or whatever swap partition might be available

cp -av /dev/null dev
```

Then set permissions appropriately:

```
chmod 666 dev/*
```

Step 8. Copy some needed files to /zipdisk/lib/

Use ldd to determine which libraries are needed for /zipdisk/lib. First for Busybox:

```
ldd /extra/bin/busybox
```

In my system this tells me that we need:

/lib/libc.so.6

/lib/ld-linux.so.2

So copy the related files e.g.

cp /lib/libc.so.6 /zipdisk/lib/

cp /lib/ld-linux.so.2 /zipdisk/lib/

Similarly, to assess what is needed for Tinylogin support enter:

ldd /extra/bin/tinylogin

In my system this tells me that I need:

/lib/libcrypt.so.1

/lib/libc.so.6

since we already have the second one we enter:

cp /lib/libcrypt.so.1 /zipdisk/lib/

For Tinylogin 0.78 we also need the Name Service Switch support by way of libnss_compat so we copy that e.g.

cp /lib/libnss_compat-2.1.3.so

Then we apply ldd to the added libraries to see if there are any deeper dependencies to find that we need libnsl. Therefore we enter

cp /lib/libnsl-2.1.3.so /zipdisk/lib

Finally we install the corresponding links and the cache via

ldconfig -r /zipdisk

Step 9. Create some needed files in /zipdisk/etc/

It may in some cases be easier to copy these from your existing /etc directory to /zipdisk/etc and then edit appropriately.

You'll need these files:

- /zipdisk/etc/fstab
- /zipdisk/etc/inittab
- /zipdisk/etc/init.d/rcS
- /zipdisk/etc/securetty
- /zipdisk/etc/nsswitch.conf
- /zipdisk/etc/ld.so.cache
- /zipdisk/etc/passwd
- /zipdisk/etc/group
- /zipdisk/etc/shadow

We'll now go through these one at a time.

/zipdisk/etc/fstab

Here is the author's fstab:

```
/dev/hdc / ext2 defaults 1 1

none /proc proc defaults 0 0

/dev/hda3 swap swap defaults 0 0
```

Keep in mind that the zip disk device, /dev/hdc, could be different for you.

/zipdisk/etc/inittab

The inittab uses slightly different conventions than usual, because it is read by the init from Busybox. See the Busybox documentation. At any rate, here it is:

```
::sysinit:/etc/init.d/rcS

::respawn:/sbin/getty 9600 -

::ctrlaltdel:/bin/umount -a -r > /dev/null 2>&1
```

/zipdisk/etc/init.d/rcS

```
#! /bin/sh

/bin/hostname zip_embed

/bin/mount -a

/bin/mount -n -o remount,rw %root% /
```

Obviously, the hostname is arbitrary. The /bin/mount line mounts everything in fstab and will complain a bit, since some are already mounted. The last line remounts the root file system.

/zipdisk/etc/securetty

```
# secure the tty's you'll use

tty1
```

/zipdisk/etc/nsswitch.conf

```
passwd:     compat

shadow:     compat

group:      compat
```

/zipdisk/etc/ld.so.cache

This was actually created at the end of step 8, by the command:

```
ldconfig -r /zipdisk
```

/zipdisk/etc/passwd

Copy this from your existing /etc/passwd i.e.

```
cp /etc/passwd /zipdisk/passwd
```

If there are entries you wish to remove, do so. For example, we removed any entries having to do with users in the /home directory, planning to later use Tinylogin's adduser once the new system was up and running. We also edited the root entry to reflect that our shell will be /bin/sh. The edited root entry looked like this:

```
root:x:0:0:root:/root:/bin/sh
```

/zipdisk/etc/group

Copy this from your existing /etc/group i.e.

```
cp /etc/group /zipdisk/group
```

If there are entries you wish to remove, do so consistent with the entries removed in the passwd file. Otherwise, no entry editing is needed.

/zipdisk/etc/shadow

Copy this from your existing /etc/shadow i.e.

```
cp /etc/shadow /zipdisk/shadow
```

If there are entries you wish to remove, do so consistent with the entries removed in the passwd file. However, we do want to edit the root (first) entry by removing the password field. For example, if the entry is something like:

```
root:sRw4kfPP!5Y15Wjk:11247:0:99999:7:::
```

then remove the field between the first and second colon, yielding something like:

```
root::11247:0:99999:7:::
```

This will allow you to login the first time as root with no password and then set one at that time with the passwd command.

Step 11. Adjusting File Permissions as needed

The appropriate file permissions are 664 for these files in /etc

- /zipdisk/etc/fstab
- /zipdisk/etc/inittab

- /zipdisk/etc/securetty
- /zipdisk/etc/passwd
- /zipdisk/etc/group
- /zipdisk/etc/shadow

and 644 for these files

- /zipdisk/etc/nsswitch.conf
- /zipdisk/etc/ld.so.cache

Finally, one level deeper we have permissions 744 for

- /zipdisk/etc/init.d/rcS

Step 10. Put the kernel image in /boot

In Section 7.4, we made a new kernel image. Copy it to your system's /boot i.e.

```
cp /usr/src/embed_linux/arch/i386/boot/bzImage /boot/emb_zip
```

Step 11. Modify /etc/lilo.conf

Add a stanza to /etc/lilo.conf along these lines:

```
image=/boot/emb_zip
  root=/dev/hdc
  label=emb
  read-only
```

where the label 'emb' will be what you enter at boot time to boot your new zip system. As before, the /dev/hdc should be replaced by your appropriate zip device name. If your system also has a boot prompt message, you might modify it to add the new label.

Step 12. Run lilo

To rewrite the master boot record so this will work run lilo at the command line:

lilo

7 - 6.3 Trying it out - Zip Disk

Reboot the system and, at the lilo prompt, choose to boot your new small Linux on the extra partition by entering:

emb<CR>

As with the floppy example, the boot process should first show on the screen a line starting with SYSLINUX giving credit to its author Peter Anvin. It then lists things discovered and accomplished in the boot process. which should then end with a login prompt. You can login as root and explore your small system.

On the other hand, if the system does not boot, go through all the steps again for another try.

7 - 7 Revisiting uClinux

It is no instructive to return to uClinux and see how it is organized and used. We'll see similarities to what we have covered in this chapter. We'll also see important differences.

Perhaps our floppy disk example is closest to the uClinux organization and usage. Our example,

- created a compressed kernel image, working in the linux source hierarchy
- created a root file system for the target in a working directory
- converted that root file system to a ROM file system and compressed it
- made a bootable floppy
- copied the compressed kernel image and compressed ROM file system to that floppy
- placed a configuration file on the floppy that guided the boot process

Then at boot the compressed ROM file system is expanded into the RAM disk.

The similar process for uClinux did these steps:

- created a compressed kernel image, working in the linux source hierarchy
- created a root file system for the target in a working directory
- converted that root file system to a ROM file system and compressed it
- concatenated the compressed kernel image and compressed ROM file system image
- loaded the concatenated image to the target's FLASH ROM

The first part of the process is quite similar. Of course, a major difference is that the uClinux target is a different architecture than the Linux development workstation. This is most noticeable by examining the uClinux source hierarchy located at /opt/uClinux/linux on the Linux workstation. For example, the directory node /opt/uClinux/linux/include/asm is linked not to the Intel node but to /opt/uClinux/linux/include/asm-m68knommu.

7 - 8 Activities/Exercises

7 - 8.1 Get Busybox

Follow Section 7 - 2.2. Don't do the installation here.

Check with your instructor, the Busybox tarball may be provided.

7 - 8.2 Create a new kernel image

Follow Section 7 - 4 carefully and create a new kernel image.

7 - 8.3 Build a Small Linux on a Floppy

Follow Section 7 - 4.2 Things needed include:

- MS DOS formatted floppy
- busybox tarball
- genromfs tarball
- syslinux tarball

Check with your instructor, these items may be provided.

7 - 8.4 Build a Small Linux on a Zip Disk or on an Extra Partition

Follow Section 7 - 5.2 Things needed include:

- A zip disk or extra partition on a hard drive
- busybox tarball
- tinylogin tarball

Check with your instructor, these items may be provided.

7 - 8.5 Investigating your zip system

Boot your zip system and login as root without a password. Then

- add a root password.
- add a new user and a user password.
- exit and login as the new user. Is the user appropriately restricted?
- try various commands.

7 - 8.6 Add networking capabilities

This is a mini project. Try adding network capabilities to the Small Linux on the zip disk. If you compiled your image without networking, you might start by reconfiguring and recompiling that image. If successful, check the size of the package. You might try it for the floppy version.

7 - 9 References

Bruce Perens, *Building Tiny Linux Systems with Busybox - Part 1*, Embedded Linux Journal, Special (Inaugural) Issue, Winter 2000.

Bruce Perens, *Building Tiny Linux Systems with Busybox - Part 2*, Embedded Linux Journal, Forthcoming.

Gerard Beekmans and Michael Peters, *Linux From Scratch*, Version 2.4, 2000. Available from http://www.linuxdoc.org/.

Sebastian Huet, *Embedded Linux Howto*, March 3, 2000. [unfinished] Available from http://www.linux-embedded.org/

Tom Fawcett, *The Linux Bootdisk HOWTO*, Version 4.1, September 2000. Available from http://www.linuxdoc.org/

CHAPTER 8

Making Your Own Embedded Linux with Embedix SDK

8 - 1 Introduction

In Chapter 7 we learned how to make an embedded kernel, using Tinylogin and Busybox. Everything we did was straightforward; however, almost everything was done via command line. In this chapter our focus remains the same. We would like to build our own small embedded kernel. In this chapter we are going to use a product produced by Lineo Inc. called Embedix SDK.

Using Embedix SDK gives us a unique advantage. Instead of using an existing kernel and removing parts we don't need, we start with a Graphical User Interface (GUI), we load a minimal base kernel which contains only required kernel code, and then we add any components that we would like. Embedix SDK is built as a GUI utilizing both Tinylogin, and Busybox.

In the previous chapter we were concerned with interdependencies and the effect on our target kernel. In this chapter we are still concerned with interdependencies; however the Embedix SDK helps us understand and control those interdependencies.

The steps and examples in this chapter are based on an ide hard drive. An example might state hda or hdb. if you have a scsi hard drive or other media, then you will need to make the appropriate adjustment.

8 - 2 Installing Caldera 2.3

By the time you have reached this section, you have already installed Linux, and you probably have it configured to your liking. Since Embedix SDK requires Caldera 2.3, and if you didn't install Caldera 2.3 as your Linux System, then you have a couple of choices. If you have one hard drive, you probably have a partition for your Microsoft Windows system, and a partition for your Linux System. The other alternative is that you have a hard drive containing only a Linux partition. We will examine both scenarios in Section 8-2.2 and Section 8-2.3. Beforehand we need to understand what we will need for partitions and hard drive space.

8 - 2.1 Requirements

In order to properly use Embedix SDK, you will need a hard drive which contains a minimum of 1.5 Gigabytes. You will also need a swap at least 64 Megabytes, but no greater than 128 Megabytes. Finally you will need a separate partition which will ultimately hold your newly created image. This partition can be 50 to 100 Megabytes. As always, before you make any changes, ensure that you have a complete backup of your crucial data.

8 - 2.2 One Hard Drive Containing Windows and Linux

If you have a hard drive which contains Microsoft Windows 98 or 95, then you can use the partition software supplied with Caldera to resize your hard drive. Boot into your Microsoft Windows system. Once this is complete place the Windows Tools & Commercial Packages compact disk into your CD ROM. On this CD is the program Partition Magic CE. This is a graphical user interface that allows you to easily resize your hard drive. Recall from the previous subsection that you will need at least 1.5 Gigabytes for the main Caldera installation, up to 128 Megabytes for a swap space, and 50 Megabytes for the installation partition. Using the Partition Magic program, provided with Caldera, allows easy changes to your partition table.

Changing your partition table can be a little tricky. Most likely your windows partition will be the primary partition with your Linux and the Linux swap space set up as logical partitions. You will just need to decrease the size of the Linux partition by approximately 50 to 100 megabytes, in order to provide the recommended partition for your image when it is created. Of course you can change your partition in many other fashions. These are outside the scope of this course, and should be discussed with your instructor.

Once your partition table is changed, you will need to follow these steps.

Step 1. Place the Linux Kernel and Installation CD into your CD ROM. This will bring up the OpenLinux 2.3 Installation dialog.

Step 2. Click Install Products.

Step 3. Click Launch Linux Install (loadlin). The program will ask you if you would like to restart in MS-DOS mode.

Step 4. Click Yes. The program will ask you to insert the OpenLinux Installation CD.

Step 5. Press the enter key. Linux will boot up to the installation screen.

Step 6. Select Language. The default is English

Step 7. Select Mouse. Default is PS/2 Note: if you select serial mouse, then you must select which COM port your mouse is on. Using a serial mouse can be very tricky, it is best to use the keyboard as much as possible, if you have a serial mouse. If you must use the serial mouse, move it very slowly.

Step 8. Select Keyboard Layout and Keyboard Language. The default layout is Generic 101 - key PC and the default language is U.S. English

Step 9. Select Video Card. Usually the Caldera 2.3 will detect the Video Card and Video RAM. These are easily changed. If your card is not automatically determined, there is a probe option.

Step 10. Select Monitor. The default is for a typical monitor of 1280x1024, 60Hz. Again, you can easily change the monitor settings to match your monitor.

Step 11. Select Video Mode. The default is 1280*1024 with a refresh rate of 87 Hz. Once you select the monitor, select the number of colors that are displayed. Then Test this mode. Note: Caldera 2.3 boots as Linux 5, which is the X Windows, KDE, graphical user interface version. It is important that you have the correct video card, monitor and video mode. Spend some time here, ensuring everything is working properly.

Step 12. Select Installation Target.

This requires these smaller steps:

Caldera 2.3 will do a scan, and give you the choice between *Entire Hard Disk*, *Prepared Partitions*, and *Custom*.

If you created your partitions with Partition Magic, then select *Prepared Partitions*.

Prepared Partitions will allow you select the appropriate partition that will contain the root file system. Note: Be careful to select the partition where your current Linux installation resides. If you select the wrong partition, you will have to start over. The partition you will select should be labeled /dev/hda? where the question mark represents a number greater than 1.

If you have a Linux only hard drive, then select the *Custom* option. The next screen will be a list of all the partitions in your computer. If you do not understand how your particular machine is configured, then note your partition table information, and consult your instructor.

If you have not created the 50 to 100 megabyte partition for hold the built image, you will need to do it here. You can delete and edit your partition table. Recall from section 8 - 2.1 you will need at least 1.5 gigabyte of space for the root file system, and the Embedix software. You will also need at least 64 megabyte for swap space. Once you have your hard drive partitioned to your liking, and the partitions meet the requirements outlined above, then click the *write* button.

Note: you do not want to select a mount point for the 50 to 100 megabyte partition. This partition will remain unmounted until the Embedix SDK software needs to mount it.

Step 13. Select Partition Information. This where you format the selected partition. You must always format. Click Format chosen partitions. Make a note of the actual partition where the swap space resides, this will be needed later for the Target Wizard program.

Step 14. Select Installation. Per the requirements of the Embedix SDK software, you must select All Packages.

Step 15. Set Root Password. The software will began installing. As it installs, you complete the rest of the installation process. Select a root password, that is at least five characters in length.

Step 16. Set Login Name(s). This is where you create a non privileged user account.

Step 17. Set Up Networking. If you do not have an ethernet card, then select No ethernet. Otherwise select *DHCP* or *statically.* If you select Ethernet configured statically, then you will need to set all the appropriate IP and DNS information. You may need your instructor's help here.

Step 18. Select Linux Loader. This is where you select the location to install LILO. Caldera 2.3 usually has a default option selected for you. You may choose this option, or you may make your own selection.

If you have Microsoft windows installed, be sure to check the box for other systems to load, or you will not be able to boot your Microsoft Windows system.

If you use software, such as Boot Magic, then do not select other operating systems to load, and change the *Where to install LILO* option to *Target partition*.

Step 19. Choose Time Zone. If you look through the options there is an option for US/Pacific. Also select if your hardware clock is set to GMT or to local time.

Step 20. Entertainment. Sit back and wait for your installation to complete. Once it is complete, click Finish, and wait for your system to boot into Linux.

8 - 2.3 Linux Only Hard Drive

Step 1. Setup your personal computer to boot via the CD ROM.

Step 2. Insert the Linux Kernel and Installation CD into your CD ROM

Step 3. Reboot your computer. Linux will boot up to the installation screen.

Step 4. Select Language. The default is English

Step 5. Select Mouse. Default is PS/2 Note: if you select serial mouse, then you must select which COM port your mouse is on. Using a serial mouse can be very tricky, it is best to use the keyboard as much as possible, if you have a serial mouse. If you must use the serial mouse, move it very slowly.

Step 6. Select Keyboard Layout and Keyboard Language. The default layout is Generic 101 - key PC and the default language is U.S. English

Step 7. Select Video Card. Usually the Caldera 2.3 will detect the Video Card and Video RAM. These are easily changed. If your card is not automatically determined, there is a probe option.

Step 8. Select Monitor. The default is for a typical monitor of 1280x1024, 60Hz. Again, you can easily change the monitor settings to match your monitor.

Step 9. Select Video Mode. The default is 1280*1024 with a refresh rate of 87 Hz. Once you select the monitor, select the number of colors that are displayed. Then Test this mode. Note: Caldera 2.3 boots as Linux 5, which is the X Windows, KDE, graphical user interface version. It is important that you have the correct video card, monitor and video mode. Spend some time here, ensuring everything is working properly.

Step 10. Select Installation Target.

This requires these smaller steps:

Caldera 2.3 will do a scan, and give you the choice between *Entire Hard Disk*, *Prepared Partitions*, and *Custom*.

If you created your partitions with Partition Magic, then select *Prepared Partitions*.

Prepared Partitions will allow you select the appropriate partition that will contain the root file system. Note: Be careful to select the partition where your current Linux installation resides. If you select the wrong partition, you will have to start over. The partition you will select should be labeled /dev/hda? where the question mark represents a number greater than 1.

If you have a Linux only hard drive, then select the *Custom* option. The next screen will be a list of all the partitions in your computer. If you do not understand how your particular machine is configured, then note your partition table information, and consult your instructor.

If you have not created the 50 to 100 megabyte partition for hold the built image, you will need to do it here. You can delete and edit your partition table. Recall from section 8 - 2.1 you will need at least 1.5 gigabyte of space for the root file system, and the Embedix software. You will also need at least 64 megabyte for swap space.

Once you have your hard drive partitioned to your liking, and the partitions meet the requirements outlined above, then click the *write* button.

Note: you do not want to select a mount point for the 50 to 100 megabyte partition. This partition will remain unmounted until the Embedix SDK software needs to mount it.

Step 11. Select Partition Information. This where you format the selected partition. You must always format. Click Format chosen partitions. Make a note of the actual partition where the swap space resides, this will be needed later for the Target Wizard program.

Step 12. Select Installation. Per the requirements of the Embedix SDK software, you must select All Packages.

Step 13. Set Root Password. The software will began installing. As it installs, you complete the rest of the installation process. Select a root password, that is at least five characters in length.

Step 14. Set Login Name(s). This is where you create a non privileged user account.

Step 15. Set Up Networking. If you do not have an ethernet card, then select No ethernet. Otherwise select *DHCP* or *statically.* If you select Ethernet configured statically, then you will need to set all the appropriate IP and DNS information. You may need your instructor's help here.

Step 16. Select Linux Loader. This is where you select the location to install LILO. Caldera 2.3 usually has a default option selected for you. You may choose this option, or you may make your own selection.

If you have Microsoft windows installed, be sure to check the box for other systems to load, or you will not be able to boot your Microsoft Windows system.

If you use software, such as Boot Magic, then do not select other operating systems to load, and change the *Where to install LILO* option to *Target partition*.

Step 17. Choose Time Zone. If you look through the options there is an option for US/Pacific. Also select if your hardware clock is set to GMT or to local time.

Step 18. Entertainment. Sit back and wait for your installation to complete. Once it is complete, click Finish, and wait for your system to boot into Linux.

8 - 3 Installing Embedix SDK

Now that your Caldera 2.3 installation is complete and you have successfully booted into your new Linux system, remove the CD from the drive and insert the Embedix SDK CD. Embedix SDK is the program that provides the GUI for us to create our own embedded Linux. Complete the following steps for this installation.

Step 1. Login as root

Step 2. Open a konsole window.

Step 3. Type mount /mnt/cdrom -o exec and press enter.

Step 4. Type cd /mnt/cdrom/embedix.sdk and press enter.

Step 5. Type ./installsdk and press enter.

Step 6. The software will install. After a few minutes, you will be prompted to install gdb debugging. Press n and enter.

Step 7. You will then be prompted to install Linux Trace Toolkit. Press n and enter.

Step 8. After your receive the message All components installed successfully. Type cd and enter.

Step 9. Type umount /mnt/cdrom and press enter.

Step 10. Open a text editor. It is probably easiest to open kedit. In the console window type kedit and press enter.

Step 11. Using kedit, open /etc/fstab

Step 12. Add this line to the end of the fstab file

/dev/hda? swap swap defaults 0 0

Where the question mark represents the number noted from section 8 - 2.4 step 8.

Save the file, and exit the kedit program.

Step 13. Close the konsole window and exit from KDE.

Step 14. Select *Shutdown*, and then *Shutdown and Restart*. Essentially reboot the system. When you system reboots you may get an error swapoff /dev/hda? invalid argument. This is acceptable, and will correct itself on the system restart.

Step 15. Boot into you Caldera 2.3 system that contains the Embedix SDK software.

8 - 4 Introduction to Target Wizard

Now that you have the software installed, and your system configured properly, we will repeat many of the same projects you considered in previous chapter, except we will allow the GUI of Embedix SDK to do all the work. As previously mentioned, the GUI is called Target Wizard. Before we create an image, you will need to understand the basics of Target Wizard.

In the previous chapter the foundation for our embedded kernel was *Tinylogin*, and *Busybox*. Again we build upon this foundation. Target Wizard was designed to aid in building an embedded Linux kernel. The natural progression is to add an installation GUI to complement *Tinylogin* and *Busybox*.

8 - 4.1 Starting Target Wizard

There are two ways to start Target Wizard. This section will mention both, and allow the user to choose a preference. First login as an unprivileged user. Everything you will need to do from this point forward, will not require super user access, the access of a normal user will be just fine. With that being said, when you installed Caldera 2.3, you were required to set up another account, besides root. Login not as root, but as that user.

If you would like to just run the program, and not see any of the calls going on, then:

press ALT and F2 at the same time.

This will bring up the XWindows command prompt. At the prompt type *tw* and press.

On the other hand, if you would like to observe the calls being produced by Target Wizard, then open a konsole window.

At the command prompt type *tw* and press enter.

Either option will activate the Target Wizard program. The first time you start the program, you will see Lineo splash screen. If you want you can deactivate this, or you can always change the splash screen to your own liking. Changing the splash screen is beyond the scope of this course; however deactivating is not.

Deactivating the splash screen is part of the options we select before we create a project.

Once the splash screen has disappeared, and the Target Wizard screen is active, you will want to verify the correct architecture and options are set. In order to do this you will need to follow these steps.

Step 1: Choose File -> Target Wizard Settings

Step 2: General Options Tab

If you do not want the splash screen to display on start up uncheck the box.

It is good practice to save your state on exit, so it is best to not uncheck this box

Step 3: Target Options Tab

Verify the target option is set to GNU/Intel 386

Step 4: Click OK

Note: It is not necessary to repeat this section every time you open the Target Wizard program. Once the changes are saved, they will remain in effect until you change them.

8 - 4.2 Creating A New Project

Once you click OK, or once you open Target Wizard, the screen will be blank. This is due to no project being created. To create a project:

Choose Project -> New

The Project Name and Location dialog box will appear.

Project Name: the name you wish to give your project. It can be any name you wish.

Project Directory: the location where your project will reside. A default directory is provided, and it is recommended that you not change this directory location.

Once you have named your project, click Next,

The Project Name and Location dialog box becomes the Directories dialog box.

Embedix Directory: the location of the Embedix SDK files. It is recommended that you not change this directory. The default is (/opt/Embedix), and there are many directories and tools under the default directory which Target Wizard requires.

Build Directory: when you build a configuration, Target Wizard saves some incremental build files in this location. If you later change your configuration, Target Wizard will not have to rebuild the files located within this directory.

Target Directory: the location of the target image, and the custom Operating System that you build

Once you have made your selection, click Next.

The Directories dialog box becomes the Build Options dialog box.

Build with conflicts: build will continue if conflicts exist. Until you better understand the conflicts that can arise, it is best to uncheck this box.

Continue building when errors occur: build will continue if there are errors. Until you better understand the conflicts that can arise, it is best to uncheck this box.

Build image after packages: build each package first, then complete the build of the target image. Leave this box checked. You will want to build the individual packages before you build the target image. This will allow you the opportunity to see if any conflicts or errors exist. Also if you make changes later on, then we will not have to rebuild these packages.

Run LIPO on image: run a library detection tool which detects the libraries and symbols required by your project. Leave this box checked.

Merge user and target images: checks the user directory for content to include in the build. Until you better understand the conflicts that can arise, it is best to uncheck this box.

Build with NFS client support: allows for NFS root kernel. Until you better understand the conflicts that can arise, it is best to uncheck this box.

It is best to review your selections to ensure everything is in order. If you followed the above recommendations, then you will have two boxes checked. The **Build image after packages** and the **Run LIPO on image** should be the only boxes that remain checked. In a later chapter you will changes these check boxes.

Once you have made and reviewed your selections, click Next.

The Build Options dialog box becomes the Target Options dialog box.

Current Project Target Platform: make a choice between the **GNU/Intel 386** or **GCU/Power PC**. This should be set to the **GNU/Intel 386** from the previous section.

Once you have made your selection, click Finish.

The Target Options dialog box will disappear. Your system will pause for a moment as Target Wizard and Embedix SDK create and populate the necessary files for your to build your embedded target image.

Target Wizard will return with the main screen now populated. This screen will have the title bar of Embedix Target Wizard: Project = “Your Project Name”: Target Platform = “GNU/Intel 386”. The Target Wizard interface will contain a dialog box/dialog menu titled Groups, Components, and Options.

You have successfully created a project. Now you are ready to start selecting the components of your embedded Linux kernel.

8 - 4.3 Loading A Kernel Configuration

Lineo has developed a set of configuration files, that encompass most embedded kernels. We will examine three of the six files in this section, and we will select the one that best meets our needs.

To load a configuration file already developed by Lineo

Choose File -> Load State

The Embedix Target Wizard: State Files dialog box will appear. There are eight choices within this dialog box.

Settop_config_PC: minimum requirements for embedded Linux on a PC.

Settop_config_MSI: minimum requirements for embedded Linux on a MSI box.

Settop_config_Elite: minimum requirements for embedded Linux on an Elite box.

PowerPC_403e_Min_NFS_Config: minimum requirements for NFS on a Power PC.

PowerPC_403e_Min_Config: minimum requirements for a Power PC.

gdb_test_config: This option depends requires the gdb installation, which we chose not to install.

Embedix_min_new: Lineo's preselected minimal Embedded Linux. This is nothing more than a system with a shell

Embedix_config_1.0: Lineo's preselected configuration which should meet typical i386 embedded Linux requirements.

The above descriptions are the general descriptions one would expect to be part of any user's manual or product description that is supplied. Before we can make a selection, we have to ask what the components really are. In other words, if we select one over another, what exact components will our Embedded Linux actually contain? In order to understand the components, it is best to do a comparison of: /

bin, /sbin, /usr/bin and /usr/sbin. This will section will do the comparison, by listing the components of each section for the three configurations of **Settop_config_PC**, **Embedix_min_new**, and **Embedix_config_1.0.**

TABLE 8 - 1 Embedix_min_new

/bin	**/sbin**	**/usr/bin**	**/usr/sbin**
login	getty	dc	fbset
ash	init	killall	
busybox	reboot	length	
cat		printf	
df		trl	
false			
kill			
ls			
mount			
ps			
pwd			
rm			
sed			
sh			
tinylogin			
true			

TABLE 8 - 2 Embedix_config_1.0

/bin	/sbin	/usr/bin	/usr/sbin
addgroup	dhcpd	basename	fbset
adduser	e2fsck	chvt	in.telnetd
delgroup	freeramdisk	clear	micro.inetd
deluser	fsck_ext2	cut	
login	fsck_minix	dc	
su	getty	dirname	
ash	halt	elvis	
busybox	init	find	
cat	ip	free	
chgrp	loadkmap	ftp	
chmod	lsmod	hostid	
chown	makedevs	killall	
cp	mkfs_minix	length	
date	mkswap	loadacm	
dd	pweroff	loadfont	
dmesg	reboot	logger	
du	rmmod	mkfifo	
echo	swapoff	nslookup	
false	swapon	passwd	
fdflush	syslogd	printf	
grep		tail	
gunzip		telnet	
gzip		tr	
hostname		tty	
kill		uptime	
ln		vi	
ls		whoami	
mkdir			
mknod			
more			

/bin	/sbin	/usr/bin	/usr/sbin
mount			
mt			
mv			
ping			
ps			
pwd			
rm			
rmdir			
sed			
sh			
sleep			
sync			
tar			
tinylogin			
true			
umount			
uname			
zcat			

TABLE 8 - 3 Settop_config_PC

/bin	/sbin	/usr/bin	/usr/sbin
addgroup	dhcpd	basename	fbset
adduser	e2fsck	chvt	in.telnetd
delgroup	freeramdisk	clear	lpd
deluser	fsck	cut	micro.inetd
login	fsck_ext2	dc	rpc.protmap
su	fsck_minix	dirname	tcpd
ash	getty	elvis	
busybox	halt	find	
cat	init	free	
chgrp	ip	ftp	
chmod	ipchains	hostid	
chown	loadkmap	killall	
cp	lsmod	length	
date	makedevs	less	
dd	mkfs_minix	loadacm	
df	mkswap	loadfont	
dmesg	pweroff	logger	
du	reboot	lp	
echo	rmmond	lpr	
false	swapoff	mkfifo	
fdflush	swapon	nc	
grep	syslogd	nslookup	
gunzip		passwd	
gzip		pbmtolj	
hostname		pgmtopbm	
kill		pnmscale	
ln		pnmtops	
ls		ppmtopgm	
mkdir		printf	
mknod		tail	

/bin	/sbin	/usr/bin	/usr/sbin
more		telnet	
mount		tr	
mt		tty	
mv		uptime	
ping		vi	
ps		whoami	
pwd			
rm			
rmdir			
sed			
sh			
sleep			
sync			
tar			
tinylogin			
true			
umount			
uname			
zcat			

By examining the tables, it is evident that the **Embedix_min_new** is simply a shell that we can easily add more functionality to. Right now this is beyond our scope. We need to understand Target Wizard, and the best way to do this is on a kernel that has more functionality. A kernel with more functionality is either the **Embedix_config_1.0** or **Settop_config_PC**. Both have the functionality that we need, with **Settop_config_PC** having more functionality. One point that probably has not been stressed enough, is the more functionality the larger the Linux kernel. The purpose is to make a minimal embedded Linux kernel. Another point to consider is what is the ultimate target for our kernel. In the previous chapter we placed the new image on a floppy, and a hard drive partition or zip disk. With these thoughts we select **Embedix_config_1.0**. It currently has all the functionality, and then some, that we need, and it will ultimately fit on a floppy.

At this point you should save your work. Anytime you make a change, add a Group, Component or Option it is suggested you save your work.

Choose Project -> Save.

8 - 4.4 Adding And Changing Components

Now that you have loaded a configuration file, it is necessary to make a few changes before you can build and install the image. This section will discuss those changes from two points of view. The first point of view is a couple of small tutorials. If this is your first time with Target Wizard, it is suggested that you work the tutorials before you build an image. The second point of view is one that considers the bare minimum required to prepare your file for building. If you are experienced with Target Wizard, and the loaded configuration meets your needs, as is, then this is the option you will want to select. In either case, both require some additional steps are required.

8 - 4.5 Target Wizard Tutorials

The first item you probably noticed was the green dots. Depending on the configuration, you may see yellow circles, green circles, or a variety of other symbols. The symbols in the Target Wizard program are very important. In order to make it easier in understanding the symbols, the help has been designed in a logical way. We will start this tutorial by opening the help window, and by leaving it displayed throughout the process.

8 - 4.5.1 Target Wizard Icon Tutorial.

Step 1: Choose Help -> Icon Legend

Step 2: Explore the contents of the *System* Group. Go down to *System* and double click. This will expand the *System* Group.

Step 3: Move down to the *Utilities* Component and double click. This will expand the *Utilities* Component.

Step 4: Move down to the Option *tinylogin* and double click. This will expand the *tinylogin* Option.

Notice the outer names represent the different Groups that are available. The next level under a Group is the Component. The level under the Component is the

Options that we are allowed to select. Options are not necessarily one line, that we can choose. As you have just seen, the *tinylogin* Option has additional Options beneath it.

Step 5: Move to *Include /bin/su?* and single click on this Option.

Step 6: Examine the *State* tab in the window below the Groups, Components, and Options. Notice the green smiley faces. These indicate a *Fulfilling Dependency*, or in other words this item is helping fulfill the node and the Option.

Notice in the *State* window, there are descriptions as to what the options requires, and if all the Options must be fulfilled before the Components and Group can be fulfilled. The object is to have all the Components, Groups, and Options for your choices, before you attempt to build the kernel.

Step 7: Move back to the top of Groups, Components, and Options. The second Group down is *Applications*. Navigate to *Applications* and double click. Notice the *Applications* group has the *Not Enabled* symbol next to it. This indicates that no Components, under it are selected. There could be Options enabled, but without the Components enabled, the Options will not be built into the image.

Step 8: Navigate to *Communication* Component and double click.

Step 9: Navigate to the *minicom* Option, and double click. Notice there are two options under *minicom*. One of the options is disabled, and one is *Enabled but Parents are not Enabled*. This indicates the Option is set to Enabled, but since the Component is not Enabled, then it will not be built into the image.

Step 10: Navigate to *Include /usr/bin/runscript and /usr/bin/ascii-xfr*?. Notice the value is set to false. This indicates that even if the Components were Enabled, then this would still not be built into the project. This is in contrast to the */usr/bin/minicom* Option which is right below it. If the Component was enabled, then this Option would be built into the image.

Step 11: Right click. A dialog box will appear. Select *Enable with Parents*. A new dialog box will appear, read it carefully and select Yes.

Step 12: This will bring up green circles indicating the Group, Component, and Options are all fulfilled, and this option will be built into the image.

Step 13: Move down to *Include /usr/bin/minicom*?. Right click and chose disable. This will bring up the Yellow Circle representing Enabled and Unfulfilled. Look down at the *State* Window below and recognize that some Options are not selected. If you want the image to build correctly, then you will need to correct the error.

Step 14: Navigate back to the *Applications* Group. Right click and select disable. This will disable this Group, Components, and all the Options. Remember this was practice to get used to the different symbols. Even though we suggest you save your work, you do not need to save your work in this case. It is actually better if you load the file you saved, before you started making changes.

Choose Project -> Open Recent Project -> Your Project Name

8 - 4.5.2 Target Wizard Conflicts Tutorial.

Step 1: Choose Help -> Icon Legend

Step 2: Navigate to the *System* Group and double click

Step 3: Navigate to the *Base* Component and double click

Step 4: Navigate to the *tar* Option and right click

Step 5: Select *Enable with Parents*. Notice the conflict symbol appears. This conflict symbol indicates this Option is in conflict with another Option of a different Component or Group, or it is in conflict with an Option in the current Component or Group.

Step 6: Scroll around among the different Components and Groups until you find the conflict.

Step 7: Navigate under the *System* Group to the *Utilities* Component and double click.

Step 8: Navigate to the Option *busybox* and double click

Step 9: Navigate to the Option *File compression utilities* and double click.

Step 10: Navigate to *Include /bin/tar*? Notice the Conflict symbol has guided you to this point. Also notice the *State* window is empty. The *State* window is meant to

guide in select all the appropriate parents and children in order to have a Group or Component that is *Enabled and Fulfilled*. This is not the case in this example.

Step 11: Select an Option to disable. In your reading from the previous chapter, and in remembering what your target is, it is better to select the option that is smaller in size. In this example that option will be the *tar* under *busybox*. Go back to the *Base* Component and disable *Include /usr/bin/tar.*

Step 12: Since this is a tutorial, it is not necessary to save your work. Normally you would probably benefit by reloading your previous saved work.

8 - 4.6 Tutorials and Target Wizard Summary

There are other conflicts and problems that may arise while working with Target Wizard. It is important that as a conflict arises, you can work through it. The ultimate goal is for any Option you select, not only is the Option *Enabled and Fulfilled*, but the Component and the Group are *Enabled and Fulfilled*. You can not build a working image until your choices are *Enabled and Fulfilled*. The best and easiest way to work through any conflicts, *Unfulfilled Dependencies*, or any other problems is by referring to the Icon Legend, which you should have displayed throughout the process.

8 - 5 Building and Installing Your Own Kernel

Now that you have worked through the tutorials you should be prepared enough to build your own image and install it. Of course this will section will walk you through step by step.

Before we start the steps to build an image, you need to ask yourself what the ultimate target is. Again, in the previous chapter you installed to a floppy and a hard drive or zip drive. For the purpose of this section, let us start with the hard drive and branch out from there.

Now that you have selected a target, the next question is which pre configured state should you load. Again, the target has to be considered. We have seen from Section 8 - 4.3 the different options for three of the pre configured states. Since we have been working on the **Embedix_config_1.0** pre configuration it is probably best to again select this configuration.

Step 1: If you have not opened Target Wizard do so now.

Step 2: Create a new project

Step 3: Load the **Embedix_config_1.0** configuration

Note steps 1 through 3 should be review, especially if you worked through the tutorials. From this point forward on into the next chapter, these three steps will be omitted. The steps will start as if the project has already been created, and some pre configuration has already been loaded.

Step 4: Navigate to the /*Embedix* Group and the *Build Image Options* Component and double click.

Notice the *Build Image* Options is *Not Enabled*. Unlike the previous chapter where you had to manually edit /etc/lilo.conf, Target Wizard will do everything for you. You will need to enable Target Wizard to edit lilo.conf, and automatically install the image.

Step 5: Navigate to *Install Target Image during Build Image* and double click.

Step 6: Navigate to *Deploy in Host Partition on Primary Drive* and double click.

Notice all the options are *Enabled but Parents are not Enabled*. Also notice the Value under the Set the Root Filesys Target Device name is set to none.

When you installed Caldera 2.3 you created an extra 50 to 100 Megabyte partition, you will need to know that partition number for the next step.

Step 7: Navigate to *Set the Root Filesys Target Device Name* and right click. This will bring up a dialog box. The set option of the dialog box is *Set Leaf Option Value*. In order to install on this extra partition, it must be unmounted.

Step 8: Select *Set Leaf Option*. At the dialog box type the address of the extra partition which you created during your installation of Caldera 2.3. The partition should be numbered hda?. Where the question mark represents a number. If it does not start with hda, then it is not the primary partition. If it starts with hdb, then it is the secondary partition, and you will need to navigate back up to I*nstall Target Image during Build Image*, and select the appropriate partition.

Step 9: Navigate back to *Deploy in Host Partition on Primary Drive w/LILO* and right click. At the dialog box, choose *Enable with Children*.

The Build Image Options should now be Enabled and Fulfilled. You are now ready to build your new image. Before you do, you should save your current project.

Choose Project -> Save.

You should also look up and down all the Groups, Components, and Options to ensure that all selected Groups are Enabled and Fulfilled.

Step 10: Click on */Embedix* and verify the *State* window indicates *Enabled, Fulfilled,* and *No Conflicts*. If all is okay, then there should be a green check mark next to each option.

Step 11: Choose Build -> Build

The *State* window will change to the *Messages* window. The image will compile and errors will be reported in the *Messages* window. If you have set everything up correctly, there should be no errors. The *Messages* window will also display the number of packages that are being built, and which package it is currently building.

Depending on you machine, the build could take some time. When the build is complete, the *Messages* window will display built ? packages of ? packages, where the question marks represent the number of packages. If you did not make any changes to the configuration, then the number of packages should be approximately 20.

8 - 6 Booting Into Your New Image

Now that you have created a new image, and Target Wizard modified lilo.conf for you, it is time to boot into your new image. Before you do, it is always good to quickly check lilo.conf ensuring all looks fine.

Step 1: Open a konsole window.

Step 2: Change to /etc

Step 3: Type less lilo.conf

Step 4: Just to be safe, check and ensure lilo.conf was edited correctly.

If lilo.conf looks in order, then close the Target Wizard program. Ensure you have saved before you close.

Log off XWindows and reboot your system.

When your system reboots, you will be given the LILO prompt. Press the tab key and type in the name of the Embedix partition. The partition should have a label similar to *embedix_hda*.

The embedded kernel will boot and the # will be displayed.

Login as root, without a password. It would be beneficial to for you to spend some time examining the different directories.

If you get the # then all was successful, If you do not get the #, then you most likely received a kernel panic message. If you receive a kernel panic, then you will need to reboot into Caldera 2.3, open Target Wizard and your project. Verify all the Groups that you selected are Enabled and Fulfilled, and rebuild your image.

8 - 7 Activities/Exercises

8 - 7.1 Install Caldera 2.3

Carefully follow Section 8 -2. If you already installed Caldera 2.3 as your main operating system, then you will be able to skip this section. When you install Caldera 2.3 don’t forget to make an additional partition.

8 -7.2 Install Embedix SDK

Carefully follow Section 8.3

8 - 7.3 Target Wizard Tutorial 1

Carefully follow Section 8 - 4.5.1.

8 - 7.4 Target Wizard Tutorial 2

Carefully follow Section 8 - 4.5.2.

8 - 7.5 Building and Installing Your New Kernel

Carefully follow Section 8 - 5.

8 - 7.6 Booting Into Your New Image

Carefully follow Section 8 - 6.

8 - 7.7 New Image Options

This is a mini project. Once you have booted into your new image try the following:

* Add a root password
* Execute a df and determine the size of the embedded image.
* Add an unprivileged user with a password
* Login as the unprivileged user and add directories
* Spend some time looking around:
  - What options do you have?
  - What options do you not have?

8 - 7.8 Different Target Wizard Options

This is a mini project. Instead of selecting **Embedix_config_1.0**, select **Settop_config_PC** and try to make the options similar to Embedix_config_1.0. Look through the options, and remove options that you feel you don't need, and add options that you would like. Build this new kernel. Once built, boot into the kernel.

8 - 7.9 Installing To a Floppy Disk

This is a mini project. One of the options under Groups Build Image Options is to Create Rescue Floppy for a Root File System. Try this or any one of the other options under Build Image Options. If successful check the size of the package.

8 - 8 References

Bruce Perens, *Building Tiny Linux Systems with Busybox - Part 1*, Embedded Linux Journal, Special (Inaugural) Issue, Winter 2000.

Bruce Perens, *Building Tiny Linux Systems with Busybox - Part 2*, Embedded Linux Journal, Forthcoming.

Embedix SDK User Guide, Lineo, Inc., October 2000. Available from http://www.lineo.com/file_index/user_manuals/index.html#embedix_sdk

Embedix Linux User Guide, Lineo, Inc., October 2000. Available from http://www.lineo.com/products/embedix/user_manual.pdf

Embedix SDK Target Wizard User Guide, Lineo, Inc., October 2000.

CHAPTER 9

Other Embedix SDK Topics

9 - 1 Introduction

In the previous chapter you experienced some of the power of Embedix SDK. You learned the basic concept of the Target Wizard Application, you built a small kernel which you hopefully installed and experimented with. In the process of building your kernel you should have experienced some small problems, possibly with hidden interdependencies. Part of the power of the Target Wizard Application is that the hidden interdependencies were handled for us, or at least we were notified of a potential problem and we were allowed to fix those problems before we built our kernel.

In this chapter, everything we learned in the previous chapter will be applied, as well as some new advanced concepts. The focus of this chapter will be to create our own customized kernel, using the configuration from the previous chapter. In the previous chapter, there were four options under the build tab which we did not use. In this chapter we examine what those options are, and actually build them into our new kernel.

In this chapter we will work with the floppy disk drive. We could use a zip disk, or another medium; however, our goal is to make a set of installation floppies, which we could use on any PC that contains a floppy drive and an extra partition.

Remembering what we learned in the previous chapter, and how powerful Target Wizard can be, we can safely work, making our advanced Embedix as painless as possible.

Before we get started, it is important to point out some of the thought processes in this chapter. Since you already have experience with Target Wizard you should already know how to change options. In this chapter, the options you need will be explained, but changing to those options will be your responsibility. This is nothing more than what was learned in the previous chapter. In this chapter we want to focus on the concepts and the ultimate goal of an embedded kernel, we are trying to not focus on the details behind the ultimate goal.

The steps and examples in this chapter are based on an ide hard drive. An example might state hda or hdb. if you have a scsi hard drive or other medium, then you will need to make the appropriate adjustment

9 - 2 Creating a Rescue Disk

In the previous chapter, one of the mini projects was to create a rescue disk. Although this seems very straight forward, there are some hidden concepts behind building the rescue disk. This section will walk you through the process of building the rescue disk, and booting into the new kernel. The concept of a rescue disk in Embedix SDK is that the rescue disk is for booting the embedded kernel. It is not meant to boot a PC that does not contain an embedded kernel.

Step 1: If you have not done so, open the Target Wizard program, and either create a new project or open an older saved project. You could load a project which you worked with in the previous chapter. We will be making changes to the *Build Image Options* Group.

Step 2: Navigate to the *Build Image Options* Group and double click.

Step 3: Navigate to the *Install Target Image during Build Image* and double click

Step 4: Navigate to *Create Rescue Floppy for a Root File System* and double click

Step 5: Choose *Target Root Device Type*. By default this option is set to IDE. This represents where the current image is located. In creating a rescue disk, Target Wiz-

ard will copy any needed files from the current working embedded image and place them on the floppy.

Step 6: Choose *Set the Root Filesys Target Device Name*. This specifies the device and partition number on the target where the Embedix image will be installed. When you boot via the rescue floppy, it will copy some of its files from the disk to this partition. This will enable you to repair any files that have lost integrity.

Step 7: You should always ensure that all build options are *Enabled and Fulfilled*. Once you have done this you can build your image. Select Build -> Build. Verify there is a floppy disk in the disk drive. Once the kernel is built, then a dialog box will appear, reminding you to place the disk in the disk drive. Select OK.

Now that you have successfully created a rescue disk, you can safely boot from that disk. Before we leave this section let us try booting with the rescue disk.

Remember this rescue disk will only work on those PC's that are set up for the *Root Filesys Target Device* which you specified. In other words, you must have the root file system on the partition before you create a rescue disk. If you do not have the root file system installed, then you will need to refer to the previous chapter.

If you were to take this to a friend's computer, and try to boot based on this rescue disk the odds are it will fail. Embedix SDK is trying to write the boot information to a specify partition which you set with the *Root Filesys Target Device* Option. If they do not have that particular partition, you will most likely receive a kernel panic. If they do have that partition, and it contains another file system, then you won't overwrite it, it will display an error concerning a mounting problem.

Note: The rescue floppy disk is not a full installation, only the Embedix Linux kernel is copied to the floppy disk.

Mark your rescue disk and set it aside. If you have future problems, then you will have a graceful way to recover.

9 - 3 Saving Your Own State

In the previous chapter, and in this chapter we most likely have been working with **Embedix_config_1.0**. If you completed the activities in the previous chapter, then created a kernel and project that is radically different from **Embedix_config_1.0**.

It would be beneficial if we could save this as a state, similar to **Embedix_config_1.0** instead of trying to remember which project we created this under. It is possible to save our own configuration as a state. This way, we can create a new project, and then just load our pre-configured kernel. For example, you may not be using dhcpd in any of your future projects, so you removed it. Another idea is you only have one additional partition setup, and you will always be writing your new image to that partition. Instead of changing the *Build Image Options* every time, you can just set it once.

Step 1: If you have not done so, open the Target Wizard program, and either create a new project or open an older saved project. You could load a project which you worked with in the previous chapter.

Step 2: Configure your project to your exact specification. Ensure that your project is *Enabled and Fulfilled*. You do not want to save your own state unless it is ready to build.

Step 3: Choose File -> Save State. This will bring up a dialog box which prompts for the Name you want for your configuration file. Type in the name you want for your configuration file.

Another option is *Storage Location*. This offers a choice between a global preconfigs directory or the local project directory. Unless you are logged in as root, the only option is Local project directory.

Once you have typed in a name and selected where to save your state, press OK.

Now that you have saved your state, you can use it anytime you want. Under File -> Load State, you will see the state that you saved. This is very beneficial, especially if you have only one other partition that you are writing to. Saving your own state will allow you to save time.

Note: When you save your state the extension sdf is appended to your configuration file name. This is saved in your projects directory, under /home/username.

9 - 4 Embedded Linux with X-Windows

Up to this point every target that has been built only operated via command line. This is acceptable for our purposes, but there may be an instance where we need a graphical user interface. This is where the power of Embedix SDK will help you. Using Target Wizard to create an image with a graphical user interface is relatively straightforward. Remember the image we are building is an embedded image. What we create with Target Wizard will not be a "full-blown" image with all the amenities that come along with a graphical user interface. It is important to always remember what the ultimate target of the embedded Linux kernel will be. Our focus has been to "keep it small." Creating an embedded Linux that contains a graphical user interface violates the "keep it small" philosophy we have maintained so far.

Step 1: If you have not done so, open the Target Wizard program, and create a new project. Now load your pre-configured state discussed in Section 9 - 3.

Step 2: Navigate down to *X11* and double click. The *X11* Option will display two choices. There first is for a smaller version of the graphical user interface, called *Microwindows* or *Nano-X*. The second is for *XFree86* version 3.3.6.

Step 3: Select the *XFree86* Option and double click. This will enable quite a few submenus. These submenus would allow you to build a very customized X-Windows.

Step 4: Navigate to the *XFree86* submenu and double click. This allows you to choose which version of XFree86 you want to install. Choices include */usr/lib/X11* or */usr/bin/X11*, as well as some others. In order to intelligently select one of these, we go back to the "keep it small" principle. Two columns over the Disk Size is displayed, it is easiest to choose an option on based on the disk size. Enable one of the sub menus.

Step 5: Once you enabled one of the sub menus from Step 4, notice the submenu *XFree86* is *Enabled and Unfulfilled*. Also notice the *X11*Group is *Enabled and Fulfilled with Unfulfilled Children*. To correct these errors, go to the *State* tab, and enable the appropriate options.

Step 6: If you have not selected a target device, do so. It is important to remember the new image will be quite large. It is recommended that you choose a partition that has approximately 100 MB.

Step 7: Verify all groups are *Enabled and Fulfilled*, then chose Build -> Build. This will began the build of your new kernel. It is very important to note that this build is quite large. You need to ensure that you have enough hard drive disk space. The build of the embedded image with a graphical user interface will require approximately 500 MB of disk space, not including the 100 MB for target installation.

Step 8: Once the build is complete, boot into your new kernel. Try to execute X-Windows, notice the differences between a full version of X-Windows and your embedded version.

9 -5 Project -> Options -> Build Options Tab

In the previous chapter, you unchecked four boxes on the Project -> Options -> Build Options Tab. We did this for two reasons. The first reason was we were very unfamiliar with the Target Wizard software. We wanted to build an embedded kernel, without having to worry about what could go wrong. The second reason was, those boxes which were deselected are advanced topic ideas in embedded Linux. In this section those boxes will be individually discussed. This discussion should give enough of a background, so that in future projects you may wish to explore some of these options.

It is also appropriate to review one of the other check boxes. We have chosen to not uncheck the box for **Run LIPO on image**. This section will explain in detail what **LIPO** is, and why this box remains checked.

9 - 5.1 Build With Conflicts

In the previous chapter, and in this chapter the focus has been to have a project that is *Enabled and Fulfilled*. Obviously, this cannot be the case if you wish to build with conflicts. The reason you would want to build with conflicts comes in when you are trying to chose between two options.

In the previous chapter, one of the tutorials covered conflicts. In that tutorial you resolved the conflict before you built your kernel. If you wanted to include both in the kernel you were creating, then you would have to enable *Build with Conflicts*.

Unless you really understand all the problems that arise, it is suggested that you keep this box unchecked.

9 - 5.2 Continue Building When Errors Occur

This allows the user to continue building, even if there is an error in the build process. This option is important if the *Build Images after Packages* is unchecked. If the *Build Images after Packages* is unchecked, and an error occurs, such as no *Target Destination* is selected, then all the packages will still be built.

This option is also important if you have selected *Build with Conflicts*. If you are purposely building with conflicts, then you are most likely going to receive errors. Obviously, this will allow your build to continue.

Unlike the build with conflicts option, *Continue Building When Errors Occur* does not cause damage. If selected *Continue Building When Errors Occur,* you will need to look through the *Messages* window to ensure your image built correctly. The *Messages* window is always a good indicator if your image had build problems.

9 - 5.3 Merge User and Target Images

This option queries the User directory for content to be included in the build. If a content is found, it is merged into the contents of the target directory, and then the target image is built.

This option is useful for those that want to add their own custom applications and then add them to the image. The creation of custom packages is outside the scope of this course; however, if you had a custom application you would need to copy your files to the appropriate directory, and you would need to enable this check box.

Example: If you had an application called foo, located in /usr/local/bin, and the configuration file in /usr/local/etc, and your project directory was /home/username/project, then you would have to copy foo from /usr/local/bin and place it in /home/username/projects/image/user/local/bin/ and the configuration file would be copied to /home/username/projects/image/user/local/etc.

This concept of merging user applications and user images is important if you have a different version of a library or a utility other than those supplied with Embedix SDK. You of course would have to copy the application or the utility to the appropriate directory, similar to what is described in the example above.

9 - 5.4 Build with NFS Client Support

This option allows you to build with a NFS root kernel. This option is important if you wish to boot your Embedix Linux image on your target using NFS root clients. Once booted with NFS, you can then test your applications. This option is outside the scope of this course. It is recommended that this box remain unchecked.

9 - 5.5 Run LIPO on Image

Target Wizard has a feature that allows you to reduce the size of the shared libraries on your system. Without reducing the shared libraries, then the target will probably not fit on a 1.4 MB floppy disk.

LIPO scans all the executable and libraries on the target device trying to determine which symbols are needed globally by the entire system. While searching, Target Wizard denotes symbols that are not used by any executable object.

It is not necessary to select LIPO; however, keep in mind the smaller the image the better. **LIPO** is a good utility, and it is best if we keep the box checked.

9 -6 Deploying A Target Image Using A Bootable Floppy Disk

Creating an embedded Linux image is not that useful if it is not portable. Since it is nearly impossible to take your development machine with you wherever you go, then there has to be an alternative. That alternative is making a set of bootable floppy disks that contain the entire embedded Linux kernel. You can than take those disks with you and install on another machine.

Step 1: If you have not done so, open the Target Wizard program, and either create a new project or open an older saved project. You could load a project which you worked with in the previous chapter.

Step 2: Navigate to *Build Image Options* and double click. Then navigate to *Install Target Image during Build Image* and double click.

Step 3: Enable the *Deployment Option from Target Using Embedix Installation Floppy.* This will bring up the *Enabled and Unfulfilled* icon. There are also quite a few subsections that you will need to choose before the *Enabled and Fulfilled* icon is provided.

Step 4: Choose the option *Target Embedix Installation*, and enter a name for the target. The default is Embedix. This is the name that will be applied to the Embedix image that is downloaded from the target.

Step 5: Choose the option *Set the Root Filesys Device name*, and enter the appropriate location of the device and partition number of where you want the Embedix image installed.

Note: This option is very important if you are going to be taking your installation disks to a different machine. Whatever you chose for the device and partition number, remember it needs to match the machine you are going to install on.

Example: You plan on installing this image on a dual boot machine, that is running windows, and does not yet have a Linux system. You will need to examine the partition table via windows to determine the partition number for the Linux system. When you have determined this device and partition information, you will need to add that device and partition to this option.

Step 6: Skip over the option *Make a Single Root Partition on Target Device for the Target Embedix Installation*. This option will create one partition on the target machine, erasing anything that may currently be installed. Since you machine is most likely configured to your liking, erasing all the data would probably not be beneficial. Ensure this option is Not Enabled.

Step 7: Choose the option *Choose Root Filesys Device Type for Target Installation*. The two choices are for IDE or SCSI. This refers to whether the target hard drive is an IDE or a SCSI hard drive. IDE is the default.

Step 8: Choose the option *Choose File System type for the Target Installation*. The two choices are EXT2 or MINIX. Since we are familiar with EXT2 from all other previous Linux installations, it is best to choose this file system type. EXT2 is the default.

Step 9: Choose the option *Create Bootable Embedix Installation Floppy Disk for Target*. This will create a bootable floppy which you can use to boot a machine and install the new Embedix image with the floppy disks.

When the machine is booted with the bootable floppy, a menu is displayed that allows the user to choose from a network installation of from the floppy disk.

Once you have enabled this option, notice the *Enabled and Unfulfilled* icon is displayed. To correct this, you will need to go to the *State* tab, and enable the appropriate options. Most likely, some networking information and some needed sockets information will not be enabled. Of course the components that are not enabled are based solely on the state that you loaded. Recall that the state used throughout this chapter is **Embedix_config_1.0**.

Step 10: Choose the option *Deployment Option 2: Create Compressed Embedix Floppy Distribution*. By selecting this option Target Wizard will prompt you to insert diskettes during the image build.

Step 11: Ensure all *Build Image* Options are *Enabled and Fulfilled*. If the *Enabled but Unfulfilled* icon appears, then you will need to go to the *State* tab, and make any changes so you receive the *Enabled and Fulfilled* icon.

Step 12: Choose Build -> Build. This will create the installation floppy. Once this is complete choose Build -> Build Image. This will write the Embedix Linux image to the diskettes. To better understand why you are executing two builds, see the note below.

Note: In order to use the *Create Bootable Embedix Installation Floppy Disk for Target option*, you will need to select this option the first time you are building the new image. After the new image is built, Target Wizard will prompt you to insert a floppy. Once the floppy is made bootable, you will need to label it as your bootable floppy, and you will need to run build again.

Now that you have created the bootable floppy disk, you will need to install the Embedix Linux image on a different floppy. To do this, you will need to disable the *Create Bootable Embedix Installation Floppy Disk for Target*. The choose Build -> Build. The new image will quickly compile. During the compile, you will be prompted to insert a floppy disk. If you new image is larger than one disk, then you will be prompted to insert another disk. As you insert and remove disks, ensure that you properly label them, their order in the installation is important.

9 - 7 Frequently Asked Questions

In writing this book, and building many different images, a few questions arose. These questions with answers were not significant enough to take a full section in any particular chapter; however, they are significant enough to at least mention.

This section will outline some of the questions that arose in producing these last two chapters, as well as what the best solution might be.

Question 1: I have an extra partition, and I would like to install multiple bootable images to this partition, is this possible?

Answer 1: No, but there are a few steps you can do to put to images on one partition. What you need to do is mount the partition, as root, outside of booting into the new image. You will then need to create a directory for your image, and move all the directories into the new image.

Now when you create a second image, it will write to the partition, but it will not overwrite your directory. LILO will be updated, and you will need to boot into the image using the target LILO created. Once you have booted, then you can navigate by using the directory you created.

Question 2: I have 2 extra partitions, is it possible to create two different images, and have LILO allow booting into either image?

Answer 2: No; however, this is being further investigated.

Question 3: I get could not find package errors while building my kernel, suggestions?

Answer 3: There are multiple steps to verify everything is configured correctly.

1) Did you build an image that was completely Enabled and Fulfilled? If you did not, then correct any disabled or unfulfilled groups and try to rebuild.

2) If all groups are Enabled and Fulfilled, then verify the swap space is correct in fstab. Refer to Section 8 - 3, Step 12.

3) If number 1 and number 2 are correct, then open a konsole window and do execute a df command. Verify the hard drive space remaining is sufficient for the options you have chosen. For example, one build started with 30% free, the option chosen was XFree86. During the build, the hard drive reached capacity, and the build was unable to complete.

Question 4: How do I clear the target partition, so I can create a new image and install to the target partition?

Answer 4: Mount the partition, outside of booting into it. For example open a konsole window, change to root and execute the command:

mount -t ext2 /dev/hda? /mnt/

Change to the partition via cd /mnt/ and execute the command rm -rf *.

Once complete umount the partition via umount /mnt/

Question 5: Where are these RPM files that are built into my image located?

Answer 5: The RPM files are located in /home/username/project/image/rpmdir. Remember every time you build a new image, Target Wizard checks this directory to see if the RPM already exists. If it does, then it uses the RPM from this directory. If it does not, then the RPM is built and added to this directory. This directory is simply a repository, building a new image does not delete the RPMs from this directory, that must be done manually.

9 - 8 Activities/Exercises

9 - 8.1 Creating A Rescue Disk

Carefully follow Section 9 - 2.

9 - 8.2 Saving Your Own State

Carefully follow Section 9 - 3.

9 - 8.3 Embedded Linux with X-Windows

Carefully follow Section 9 - 4.

9 - 8.4 Deploy A Target Image Using A Bootable Floppy Disk

Carefully follow Section 9 - 7

9 - 8.5 Different Target Wizard/X-Windows Options

This is a mini project. Build a new image that contains either X-Windows or Nano-X. Try to configure the options, so it close to a “normal” graphical user interface. Once the build is complete, boot into the new image, and open X-Windows. Configure your image from within this X-Windows session. For example: try changing the password, or adding an unprivileged user account.

9 - 8.6 Minimal Embedded Image

This is a mini project. Create a new project, load the **Embedix_min_new** state. Remove any options that you will not need. The focus is to try to make an image that is as small as possible and still functional. Once your configuration is complete, select your target as /dev/fd0, or possibly a zip drive. You will need to verify the location of your zip drive, for example, it might be hdc or hdd.

9 - 9 References

Embedix SDK User Guide, Lineo, Inc., October 2000. Available from http://www.lineo.com/file_index/user_manuals/index.html#embedix_sdk

Embedix Linux User Guide, Lineo, Inc., October 2000. Available from http://www.lineo.com/products/embedix/user_manual.pdf

Embedix SDK Target Wizard User Guide, Lineo, Inc., October 2000.